1982
APL Users Meeting

An Introduction to APL for Statisticians and Economists

Sponsored by:
I.P. Sharp Associates Limited

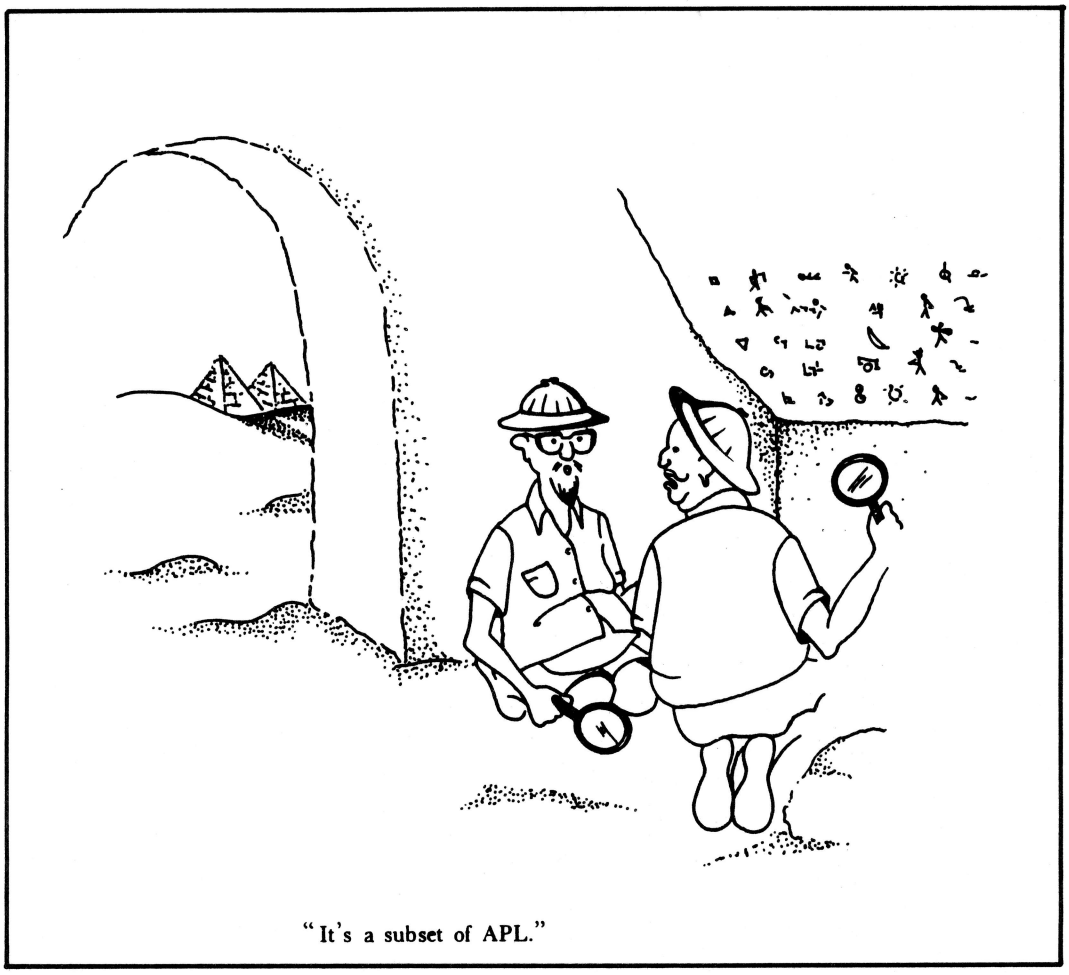
1982 APL USERS MEETING

TUTORIAL

AN INTRODUCTION TO APL FOR STATISTICIANS AND ECONOMISTS

Andrew North
I.P. Sharp Associates Limited
Ottawa, Ontario

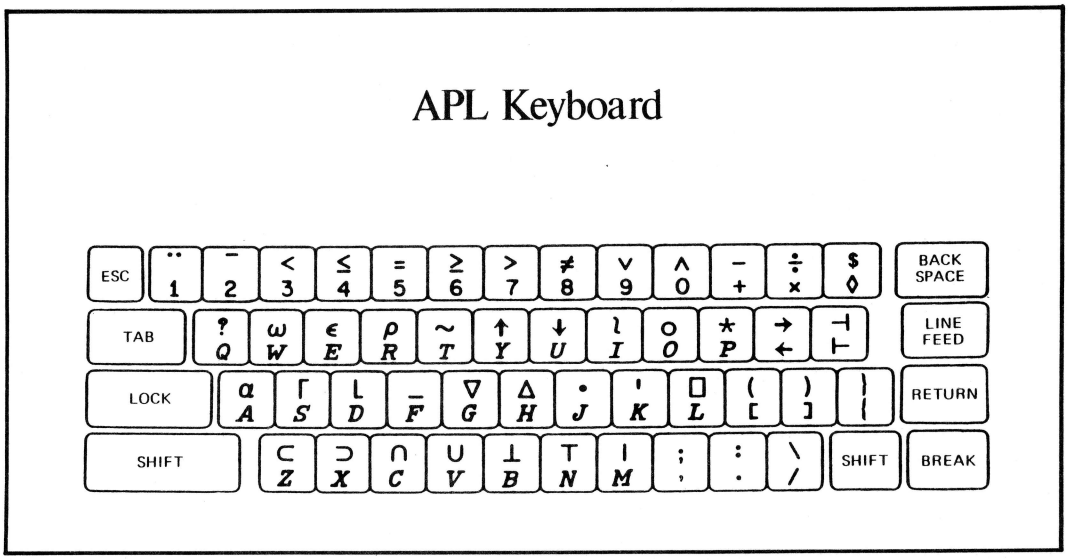
Printed in Canada
October 1982
Publication Code: 0186 8210 E1
ISBN: 0-86493-072-0



"It's a subset of APL."

This cartoon is well-known within the APL community. The reason behind its renown is that it propagates what those familiar with APL know to be a myth: that APL consists of a bewildering array of obscure hieroglyphics. This tutorial should demonstrate that this is not so. APL's notation is concise, at times even elegant, and faithful to the fundamental mathematical concepts it represents.

The APL keyboard



This is the layout of the APL keyboard. You'll notice that the lower case (non-shift) characters, representing the letters A through Z and the ten numeric digits, occupy their usual typewriter positions. These characters are used in the construction of the names of APL objects, as we'll see in a moment. The upper case, or shift, characters represent many of the primitive functions of the language. (A primitive function is a pre-defined function, such as + or -, that APL provides.) A number of these characters will be familiar to you, and the purposes served by them evident; others you've likely never seen before. Additional primitive functions, not shown directly on the keyboard, can be constructed by overstriking selected pairs of individual functions. In all there are some sixty or more different primitive functions in the language.

The syntax of APL

APL was devised originally by a mathematician as a means of communicating complex, rigorous procedures between one person and another (as distinct from between person and machine). Given the precise nature of the procedures to be communicated, it's not surprising that what resulted was a notation very similar to that of mathematics.

Most APL primitives are counterparts of common mathematical functions; the purposes served by these primitives are those fundamental to the computational requirements of the practicing statistician or econometrician.

Consider first an APL program which calculates the arithmetic mean of a string of numbers. We could calculate the average score of nine students on a math test, say, by typing:

```
      MEAN 66 82 86 57 51 74 69 85 60
70
```

Note that what you, the user, type is always indented six spaces, i.e. the carriage automatically moves six spaces to the right to await your next input. Results generated by the computer are printed or displayed on your terminal starting at the far left margin. User input is therefore readily distinguishable from output.

In APL, a program such as *MEAN* is called a **function**. Rather than typing the nine test scores explicitly every time we need to do anything with them, we can define a **variable** which contains them:

```
      SCORES←66 82 86 57 51 74 69 85 60
```

A variable is defined using the left-pointing arrow ←. The name to the left of the arrow is specified as consisting of those values to the right of the arrow. To display the contents of a variable, simply type its name:

```
      SCORES
66 82 86 57 51 74 69 85 60
```

Simple analysis of the test results can then be performed by referring to the variable *SCORES*:

```
      MEAN SCORES
70
      VARIANCE 66 82 86 57 51 74 69 85 60
161
      VARIANCE SCORES
161
      SDEV SCORES
12.688577
```

Note that none of the functions *MEAN*, *VARIANCE* and *SDEV* is a primitive APL function. You won't find them anywhere on the APL keyboard. Each is instead what is called a **defined function**. A defined function is a program written by an APL user, employing the APL primitive functions as tools. Once written, a user-defined function becomes itself an additional tool, and can be used in place of a more complicated expression, as we'll see in a moment.

Let's have a look at some simple primitive functions, beginning with addition and subtraction (recall that a primitive function is one provided by APL, and always available to an APL user):

```
      2+2
4
```

An encouraging result.

```
      5-1
4
      2+10 20 30 40
12 22 32 42
      31-0 3 0 1 0 1 0 0 1 0 1 0
31 28 31 30 31 30 31 31 30 31 30 31
```

Clearly, addition and subtraction can apply between single numbers, or between a single number and a string of numbers. We can also add/subtract one string of numbers to/from another string:

```
      10 20 30+1 2 3
11 22 33
      11 12-1 2
10 10
```

Or, these strings could be variables:

```
      10+SCORES
76 92 96 67 61 84 79 95 70
      SCORES-SCORES
0 0 0 0 0 0 0 0 0
```

But suppose we were to attempt to add together two strings of unequal length:

```
      SCORES+5 5 5 5 6 6 6 6
LENGTH ERROR
      SCORES+ 5 5 5 5 6 6 6 6
      ^
```

The variable *SCORES* contains nine elements. APL is uncertain as to how to go about adding eight elements to nine, and decides it is unable to do so. The resulting error

message tells us what the problem is (the two strings being added together are of inconsistent lengths), and the caret (^) tells us where the problem arises. It is important to note that when an error is encountered, it is reported immediately, and in a helpful and meaningful manner. You need not fear making an error, because you can rectify things immediately if you do. In the situation above, having been made aware of our mistake, we could simply retype the statement, this time including the ninth number we omitted originally:

```
      SCORES+5 5 5 5 6 6 6 6 7
71 87 91 62 57 80 75 91 67
```

Errors in defined functions are treated in the same way. If you make a mistake in your program, the error will be reported as soon as you execute that program. You can then correct the error immediately, and resume execution from the point at which the error occurred, rather than starting all over again. APL's treatment of errors, both for primitive functions and for defined functions, is one of the language's most convenient features.

* * *

Two other common primitive functions are multiplication and division. They apply to single numbers or to strings of numbers, as do addition and subtraction:

```
      4×6
24
      SCORES÷100
0.66 0.82 0.86 0.57 0.51 0.74 0.69 0.85 0.6
      12+8÷4
14
```

A question arises here. Which arithmetic operation is performed first? Are 12 and 8 added, and their sum divided by 4? Or is 8 divided by 4, and 12 then added to the quotient? One way of resolving this question would be to decree arbitrarily that certain operations are to be performed before certain others, which are themselves to be performed before still others, and so on. That is, impose a hierarchical order of execution. Some languages do so. The richness of APL precludes such a hierarchical structure, however; having to remember a hierarchy of more than 60 functions is an unreasonable burden to place upon a simple programmer. Instead, APL applies a single rule: *APL always executes from right to left.* Hence the result 14, rather than 5, above.

You can dictate the sequence in which operations are performed by using parentheses. Use of parentheses is often required when APL's right-to-left execution rule would otherwise produce a result different from that desired. APL applies the right-to-left rule within any pair of parentheses, then considers the result of the expression within the parentheses a single object, and applies the right-to-left rule again. In the expression above, we could ensure that the addition be performed before the division by using parentheses, as follows:

```
      (12+8)÷4
5
```

Or, we could obtain the same result without using parentheses:

```
      .25×12+8
5
```

Redundant parentheses, as in

```
      12+(8÷4)
14
```

are acceptable, and can always be included to improve readability.

* * *

As an illustration of the use of defined functions as additional tools, recall the defined functions named *VARIANCE* and *SDEV*. Note that, since the calculation of the standard deviation requires the prior calculation of the variance, the function *VARIANCE* can be used within the function *SDEV*. The actual APL expression which calculates the variance need not be repeated. The defined function *VARIANCE* can be used in its place:

```
      VAR←VARIANCE SCORES
      VAR
161
      VAR*.5
12.688577
```

The variable *VAR* has been defined as the variance of *SCORES*. Again, to display *VAR* we type its name. The primitive function * raises the variable or constant on its left to the power on its right. Thus, the expression *VAR**.5 raises the value of *VAR* (161) to the power ½, producing the square root of *VAR*. This result is the same as that produced by the defined function *SDEV*.

The standard deviation could instead be calculated directly (in a single step) from the variance by means of the expression

```
      (VARIANCE SCORES)*.5
12.688577
```

The parentheses dictate that the variance is to be calculated first, and the square root then applied to the result. If the parentheses had not been included, APL would have calculated the square root of all nine test scores (*SCORES*⋆.5) and then applied the *VARIANCE* function to these nine transformed scores:

```
      VARIANCE SCORES⋆.5
0.5859009806
```

This would have been quite legitimate as far as APL is concerned, but would not have produced the result you really wanted.

The use of defined functions as additional tools is further illustrated as we calculate the range of test scores:

```
      MAXIMUM SCORES
86
      MINIMUM SCORES
51
      RANGE SCORES
35
```

Because the range is simply the difference between the maximum and minimum values, we can use the two functions *MAXIMUM* and *MINIMUM* in the range calculation:

```
      (MAXIMUM SCORES)-MINIMUM SCORES
35
```

Suppose now that we need to prepare a report summarizing our somewhat superficial analysis of the test scores. We wish to present the numeric results, with comment, but the six decimal places provided by the function *SDEV* are more than are required. The defined function *ROUND* can be used to round off our results:

```
      SDEV SCORES
12.688577
      0 ROUND SDEV SCORES
13
      2 ROUND SDEV SCORES
12.69
```

The number to the left of the function *ROUND* specifies the number of decimal places to which the result (or results) to the right of *ROUND* are to be rounded.

There are two fundamental features of APL syntax illustrated here. The first is that the right-to-left order of execution that applies to APL primitive functions applies in exactly the same way to defined functions as well. In the examples above, APL first calculates the standard deviation of the variable *SCORES*, then applies the function *ROUND* (to either 0 or 2 decimal places) to this result. The flow of execution moves ever leftward, an intermediate result (not printed) being generated after *SDEV* has

completed execution, but before *ROUND* begins. As long as they return a result (more on that later), defined functions can be strung together in a sentence-like sequence.

The second important feature to note is that some defined functions refer to a single object to the right of the function name, while others refer to objects both to the left and to the right of the function name. The objects to the left (if any) and right of the name of a defined function are called the **arguments** of that function. A function which takes a single argument is called a **monadic** function, and one that takes two arguments a **dyadic** function. The single argument to a monadic function always follows the function name. The functions *MEAN*, *VARIANCE*, *SDEV*, *MAXIMUM*, *MINIMUM* and *RANGE* above are monadic functions. *ROUND* is a dyadic function. Note that arguments are in effect a means of passing information, such as data or parameters, to a defined function. There do exist defined functions (called **niladic** functions) which take no arguments; passing information to such functions must be accomplished by some other means. We won't consider niladic functions further here.

APL primitive functions follow these same syntactic rules. An APL primitive function takes either one or two arguments, and is referred to as a monadic or dyadic function accordingly. Many functions can be used both monadically and dyadically, serving slightly different purposes depending on the number of arguments. Let's examine briefly some simple but useful monadic and dyadic functions, then return to our defined functions *MEAN* and the rest to see how the primitive functions are used within the defined functions.

Common APL primitive functions

The functions + and × perform dyadically as illustrated above. + adds one or more numbers to one or more numbers, while × multiplies one or more numbers by one or more numbers. The function - can be used monadically to change the sign of one or more numbers:

```
      -6 0 ¯7.4 3.5 ¯2
¯6 0 7.4 ¯3.5 2
```

The proper name for this function is **algebraic negation**, but it is more commonly referred to as **negative**. - can also be used dyadically (in which case it is called **minus**) to subtract one or more numbers from one or more numbers:

```
      3 5 7 ¯4-8 5 0 ¯3
¯5 0 7 ¯1
```

Note the distinction between the function - (shift + on the keyboard) and the symbol ¯ (shift 2). The latter is called **high minus**, and simply indicates that a number is negative. It is not a function; it is merely notation.

The dyadic function ÷ (shift ×) divides one or more numbers on its left by one or more numbers on its right, as we've seen. With no left argument, ÷ is called **reciprocal**, and calculates the reciprocal of all numbers which follow it. For example:

```
      ÷1 ¯2 4 10 ¯12
1 ¯0.5 0.25 0.1 ¯0.08333333333
      3 ¯6 9÷6
0.5 ¯1 1.5
```

Note that the monadic use of ÷ is in effect a special case, namely 1÷*X*, of the dyadic use.

We saw earlier an example of the dyadic use of ⋆ (shift *P*), the **power** function. The one or more numbers in the left argument are a base (or bases) to be raised to the one or more powers in the right argument. The monadic function ⋆, called **exponential**, raises the constant *e* (the natural logarithm base) to the power or powers given in the right argument:

```
      ⋆0 1 2 .5
1 2.718281828 7.389056099 1.648721271
      2⋆¯1 ¯.5 0 .5 1 2
0.5 0.7071067812 1 1.414213562 2 4
```

⋆1 gives the value of the natural constant *e*. Again, the monadic use of ⋆ is a special case of its dyadic use.

The monadic function ⍟ (constructed by overstriking ⋆ with ○, shift *O*) calculates the natural logarithm of its argument. When the function has a left argument as well, the left argument is considered the logarithmic base:

```
      ⍟2.718281828 7.389056099
1 2
      2⍟1 2 4 8 64
0 1 2 3 6
      ⍟⋆1
1
```

For positive right arguments, ⋆ and ⍟ are inverse functions. Thus, ⍟⋆*X* and ⋆⍟*X* are equivalent to *X* for *X*>0. Once again, the monadic form of ⍟ is a special case of the dyadic form. It is becoming apparent that primitive functions which can be used both monadically and dyadically have dyadic forms which are generalized extensions of their monadic form.

The monadic functions ⌊ and ⌈ (shift *D* and *S*, called **floor** and **ceiling**) produce integer results by rounding their arguments down and up respectively. ⌊ generates the largest integer not greater than the value or values of the argument, and ⌈ the smallest integer not less than the value or values of the argument. ⌊ and ⌈ have dyadic counterparts as well, called **minimum** and **maximum**. They produce the minimum and maximum values, respectively, of corresponding pairs of numbers in the left and right arguments. For example:

```
      ⌊1.5 4.99 5 0 ¯1.2
1 4 5 0 ¯2
      ⌈¯3 6.7 4.01 ¯2.7 9
¯3 7 5 ¯2 9
      41 66 52⌊38 55 77
38 55 52
      0⌈10 ¯6 4 0 ¯1
10 0 4 0 0
```

The latter is useful as a convenient means of replacing all negative values by zero.

The monadic function | (shift *M*) is properly called **magnitude**, but is more commonly known (for real numbers) as **absolute value**. Every negative number in its argument is made positive, so that all elements of the result are positive or zero. Its dyadic counterpart is called **residue** or **remainder**, and produces the amount left over when the left argument is divided into the right as many integer times as possible. For example:

```
      |22 4 ¯7 0 ¯1.6
22 4 7 0 1.6
      2|71 74 74 69 70
1 0 0 1 0
```

Note that this dyadic example produces a Boolean string, or sequence of logical results, indicating which numbers in the right argument are even and which are odd. We'll see more of these Boolean variables later on. You might recognize that the residue function represents the algebraic concept of modulus of a relation.

Before we examine any more new functions, let's reconsider the user-defined functions *MEAN*, *VARIANCE*, etc., and attempt to construct the actual APL expressions that these defined functions represent. What arithmetic operations need we perform to calculate the mean of our test scores? We need the sum of the scores, and the number of scores. We could obtain the sum using the dyadic function + between successive elements of the variable *SCORES*:

```
      SCORES
66 82 86 57 51 74 69 85 60
      66+82+86+57+51+74+69+85+60
630
```

Note that because of the right-to-left execution rule, 85 is first added to 60, 69 then added to their sum, and so on. Clearly, this could become tedious (suppose we were dealing with 500 scores!). It would be convenient if there were some means of indicating that the same single function were to apply to successive pairs of elements of a variable. There is. The means is called **reduction**:

```
      +/SCORES
630
```

The / with a variable name following and a function preceding results in the application of the function in pairwise fashion to all elements of the variable. You can think of reduction as simply inserting the function (in our case, +) in between each pair of elements in the variable, and executing the resulting APL expression. Our example is called **plus-reduce**, and is clearly equivalent to the everyday operation of summation, i.e.

$$+/X \leftrightarrow \sum_{i=1}^{n} x_i$$

The function preceding the / need not be +. A number of other primitive functions could be supplied. Two of the more common are ×/ and ⌈/. The former inserts the function × between successive pairs of elements of a variable. It's called **times-reduce**, and is equivalent to the common product:

$$\times/X \leftrightarrow \prod_{i=1}^{n} x_i$$

The latter is called **max-reduce**, and produces the largest value in a variable.

The / itself is what is called an **operator** in APL. An operator is not a function, but rather takes a function (or functions) as an argument (or arguments). A monadic operator combines with a function, as in +/, to form a **derived function**, which is then applied to the data to its right. In the case of +/*SCORES*, the operator / takes as its left argument the primitive function + ; the resulting derived function +/ takes as its right argument the variable *SCORES*. / is called reduction because it reduces by one the rank of the variable to which its left argument applies (more on rank when we look at arrays later on). Our plus-reduce example reduces a string of numbers to a single number.

There's our sum. Now we need to know how many individual constituents there are to that sum (note that two of the great conveniences of reduction are that it applies to strings of any length, and that you need not know this length beforehand). The monadic function **shape**, denoted by the symbol ⍴ (shift *R*) produces the shape of its argument. When the argument is a string of numbers, as it is in our case, the "shape" is the number of elements in the string:

```
      ⍴SCORES
9
```

So, all we need to generate the mean score is the ratio of the sum of the scores to the number of scores, i.e.:

```
      (+/SCORES)÷⍴SCORES
70
```

The parentheses are needed to ensure that the +/ is executed before the division is (although note that

$$(+/X)\div\rho X \quad \leftrightarrow \quad \frac{1}{n}\sum_{i=1}^{n} x_i$$

and

$$+/X\div\rho X \quad \leftrightarrow \quad \sum_{i=1}^{n}\frac{x_i}{n}$$

so that the two happen to be equivalent). Thus, the defined function *MEAN* consists simply of a single APL expression of the form:

```
      (+/SCORES)÷⍴SCORES
```

We'll see exactly what *MEAN* looks like in just a moment, but let's deal first with the expressions that constitute the rest of our defined functions.

To calculate the variance of the test scores, we first need to transform the scores into deviations from their arithmetic mean. Since we already have this mean, a dyadic - suffices:

```
      SCORES-70
¯4 12 16 ¯13 ¯19 4 ¯1 15 ¯10
      SCORES-MEAN SCORES
¯4 12 16 ¯13 ¯19 4 ¯1 15 ¯10
```

Next, we need to square the deviations and sum the resulting squares:

```
      (SCORES-MEAN SCORES)*2
16 144 256 169 361 16 1 225 100
      +/(SCORES-MEAN SCORES)*2
1288
```

Parentheses are essential here. Without them, we would be squaring the nine individual scores, calculating the mean of these squared values, and subtracting this mean from the raw scores. An intriguing approach, perhaps, but hardly what we'd expect in a variance calculation. The derived function +/ reduces the nine squared deviations to the single number which is their sum. Finally, we need to divide this sum by one less than the number n of test scores:

```
      (+/(SCORES-MEAN SCORES)*2)÷8
161
      (+/(SCORES-MEAN SCORES)*2)÷¯1+⍴SCORES
161
```

We know that n-1 equals 8 in our case, but replacing the explicit 8 by the more general ¯1+⍴*SCORES* renders the expression usable for strings of any length. Once again, it is clear that the outer pair of parentheses is necessary, given the right-to-left execution rule. The defined function *VARIANCE*, then, consists of a single expression similar to the second expression above. Given this function, we saw earlier that the defined function *SDEV* need consist only of an expression of the form:

```
      (VARIANCE SCORES)*.5
12.688577
```

What about *MAXIMUM* and *MINIMUM*? Reduction is all we need here:

```
      ⌈/SCORES
86
      ⌊/SCORES
51
```

The variable *SCORES* is reduced to a single number by the application of the dyadic functions ⌈ and ⌊ respectively to successive pairs of test scores. The defined function *RANGE* then produces the simple difference between the maximum and the minimum:

```
      (MAXIMUM SCORES)-MINIMUM SCORES
35
```

Or, more directly:

```
      (⌈/SCORES)-⌊/SCORES
35
```

Finally, consider our function *ROUND*. The six defined functions that we have just discussed all consist of a single APL expression. Can the same be said of *ROUND* (which is dyadic, remember)? Yes, but the expression is not as obvious as those we've seen so far. We should somehow be able to use floor or ceiling to perform rounding:

```
      SIGMA←SDEV SCORES
      SIGMA
12.688577
      ⌊SIGMA
12
      ⌈SIGMA
13
```

Neither ⌊ nor ⌈ alone is sufficient, since the former always rounds down, the latter up. Note, however, that if .5 is added to *SIGMA*, and then ⌊ is applied, the result will be ⌊*SIGMA* for those values of *SIGMA* which should be rounded down, and ⌈*SIGMA* for those which should be rounded up:

```
      ⌊.5+3.1 3.7 3.99 4 ¯2.1 ¯2.9
3 4 4 4 ¯3 ¯3
      ⌊.5+SIGMA
13
```

This expression now rounds in the appropriate direction, but always to the nearest integer, i.e. always to 0 decimal places. We can use the same algorithm to round to any given number of decimal places if we first scale the data upwards, then apply the algorithm, and scale the data back down afterwards. The scale factor should be a power of 10 equal to the number of decimal places required. For example, to round *SIGMA* to two decimal places:

```
      SIGMA×100
1268.8577
      ⌊.5+SIGMA×100
1269
      .01×⌊.5+SIGMA×100
12.69
```

In general, to round a number to n decimal places, we need to complete three steps in sequence:

1. scale the number upwards by 10 to the n
2. apply the simple rounding algorithm
3. scale the number downwards by 10 to the n.

Given a value for n, this three-step procedure can be accomplished using the single expression:

```
(÷10*N)×⌊.5+SIGMA×10*N
```

For example:

```
      N←1
      (÷10*N)×⌊.5+SIGMA×10*N
12.7
```

The function *ROUND* consists of just such an expression. Before we leave *ROUND*, suppose we wish to calculate and display the test scores after standardization. We need to subtract from each score the mean score, and divide each of these differences by the standard deviation of the scores:

```
      STANDARDS←(SCORES-MEAN SCORES)÷SIGMA
      STANDARDS
¯0.3152441625 0.9457324875 1.26097665 ¯1.024543528 ¯1.497409772
       0.3152441625 ¯0.07881104062 1.182165609 ¯0.7881104062
      2 ROUND STANDARDS
¯0.32 0.95 1.26 ¯1.02 ¯1.5 0.32 ¯0.08 1.18 ¯0.79
```

Clearly, *ROUND* applies equally well to strings of numbers as it does to a single number. We've already seen numerous examples of the same primitive or defined function applied successfully both to one number and to many numbers. When we discuss arrays, you'll see that a function can apply simultaneously to all elements of an array, regardless of how they might be arranged.

Function definition

Every APL expression we've seen so far has executed in what is called **immediate execution mode**. You type the expression, and that expression performs its duties as soon as you press RETURN. Whether anything prints or displays on the terminal depends on whether you assign the result of the expression to a variable name. Our expressions have involved both primitive functions and defined functions. But how do we go about defining one of these defined functions? How do we write an APL program?

The symbol ∇ (shift *G*), called **del**, signals to APL that you wish to define a function, rather than execute an expression directly. ∇ is the symbol of transition between immediate execution mode and what is called **function definition mode**. Within function definition mode, an expression that you type is not executed; instead it becomes one of one or more statements which in sequence constitute the defined function.

∇ must be followed on the same line by the **function header**. The function header contains three vital pieces of information, the most obvious being the name of the function. More on the function header in a moment. Once you've supplied the function header and pressed RETURN, APL prompts you for the program's first statement (APL expression). This prompt takes the form of a line number, starting with 1, enclosed in square brackets. You enter the first statement of your program following the bracketed line number, and press RETURN again. The next line number in brackets then serves as a prompt for another statement. When you have finished entering statements, type another ∇. The second ∇ closes function definition, and returns you to immediate execution mode. Defining a function, then, is accomplished as follows:

```
      ∇ function header    RETURN
[1]     first statement in your program    RETURN
[2]     second statement in your program    RETURN
 .                .              RETURN
 .                .              RETURN
 .                .              RETURN
[n]     nth statement in your program    RETURN
[n+1] ∇    RETURN
```

APL produces the bracketed line numbers on the left; you type the rest.

Now, what does the function header look like? The three vital pieces of information it provides are:

1. the name of the function.

2. the number of arguments (0, 1 or 2) the function is to take. This classifies the defined function as niladic, monadic, or dyadic respectively.

3. whether the function returns an explicit result. If the result of a function's execution is to be referred to subsequently (for example, assigned to a variable or acted upon by another primitive or defined function), then that function must return an explicit result. No explicit result need be returned if, for example, the function simply displays some calculated results not required thereafter.

As an example, consider our better-than-average function *MEAN*:

```
      ∇R←MEAN X
[1]   R←(+/X)÷⍴X
      ∇
```

The function header obviously contains the name of the function. *MEAN* is monadic, which means its single argument follows the function name. Why is this argument named *X* in the function header, when we called it *SCORES* in our analysis of test scores? You'll recall that we saw earlier that arguments were in essence a means of passing information to a function. Note that line 1 of the function, in which the actual

calculation is performed, refers to a variable *X*. When you execute the function *MEAN*, the variable name that you supply as the argument, whatever it may be (*SCORES* in our case), is *always* passed via the function header to line 1 as the variable *X*. This means, of course, that line 1 is able to execute successfully regardless of what variable name you supply. It's not necessary that the argument name you specify when you type the function header be *X*; what is necessary is that this argument name be identical to that referred to in line 1 (or in subsequent lines).

The ← in the function header indicates that a result is returned by the function. The result of the mean calculation is assigned (temporarily) to a variable named *R*. This result can then be passed along as an argument to another primitive or defined function. The important consequence of *MEAN*'s returning an explicit result is that the result returned can be referred to subsequently. We did so, in fact, when we calculated

```
      SCORES-MEAN SCORES
```

Note that no variable named *R* exists once execution of *MEAN* is complete; in the expression above the value of *R* is passed along as the right argument of the dyadic function - . If we wish to assign the result of *MEAN* to a variable, we can do so:

```
      XBAR←MEAN SCORES
```

(or *R*←*MEAN SCORES*, for that matter).

If *R*← were not included in the function header (and were not present at line 1), we could still execute *MEAN SCORES* and obtain the printed result 70. Subsequent reference to this result, however, would no longer be possible. For example:

```
      ∇MEAN1 A
[1]   (+/A)÷⍴A
      ∇

      MEAN SCORES
70
      MEAN1 SCORES
70
      3+MEAN SCORES
73
      3+MEAN1 SCORES
73
RESULT ERROR
      3+MEAN1 SCORES
        ∧
```

APL reminds us that *MEAN*1 must return an explicit result so that this result can constitute the right argument expected by the dyadic function + .

Our six other defined functions look like this:

```
      ∇R←VARIANCE X
[1]   R←(+/(X-MEAN X)*2)÷¯1+⍴X
      ∇

      ∇Z←SDEV A
[1]   Z←(VARIANCE A)*.5   or   [1]   Z←((+/(A-MEAN A)*2)÷¯1+⍴A)*.5
      ∇

      ∇R←MINIMUM B
[1]   R←⌊/B
      ∇

      ∇Z←MAXIMUM X
[1]   Z←⌈/X
      ∇

      ∇R←RANGE A
[1]   R←(MAXIMUM A)-MINIMUM A   or   [1]   R←(⌈/A)-⌊/A
      ∇

      ∇Z←N ROUND B
[1]   Z←(÷10*N)×⌊.5+B×10*N
      ∇
```

The right argument to each of these functions is called either *X*, *A* or *B*. Any name can be used, as long as the argument name given in the function header matches that used to refer to the same variable within the function. The function *ROUND* takes two arguments: the right argument *B* represents the number or numbers to be rounded, while the left argument *N* specifies the number of decimal places to which *B* is to be rounded. Note that *N* and *B* serve exactly the same purpose: as arguments, each is a vehicle by which information can be passed from the user to the program.

All six functions return an explicit result, as indicated by the ← in each function header. The name to which the result is temporarily assigned need not be anything in particular. Because they return explicit results, these functions can be strung together, as we saw earlier:

```
      2 ROUND SDEV SCORES
12.69
```

The monadic function *SDEV* executes first, and its explicit result (`12.688577`) is then passed along as the right argument to the dyadic function *ROUND*.

As you might expect, there exist facilities by which we can edit defined functions (as well as expressions for immediate execution). These facilities will not be discussed here.

Before we abandon function definition to examine some other useful primitives, an exercise: write a function named *CORRELATE* to calculate the coefficient of simple correlation between two strings of numbers (strings of equal length). Assume for now that we only need to calculate and display the coefficient. Therefore, we do not need an explicit result. We as users must provide the program with two strings of numbers before it can perform the calculations. Two arguments would therefore be convenient. Our decisions regarding the structure of the function header, then, have been made. We want a dyadic function, with no explicit result, named *CORRELATE*:

```
∇X CORRELATE Y
```

Within the body of the program we need to calculate r as given by:

$$r = \frac{\sum_{i=1}^{n} (x_i - \bar{x})\ (y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n} (x_i - \bar{x})^2 \sum_{i=1}^{n} (y_i - \bar{y})^2}}$$

The first step in the computation is to transform the two variables into deviations from their respective means. We know how to do this:

```
XDEV←X-(+/X)÷⍴X
YDEV←Y-(+/Y)÷⍴Y
```

The numerator of r is then the sum of the products of these two new variables:

```
NUM←+/XDEV×YDEV
```

To compute the denominator we need to multiply the sum of squares of *XDEV* by the sum of squares of *YDEV*, then take the square root of the resulting product:

```
DEN←((+/XDEV*2)×+/YDEV*2)*.5
```

The coefficient r of simple correlation is then the ratio of *NUM* to *DEN*:

```
NUM÷DEN
```

The complete function *CORRELATE* looks like this:

```
      ∇X CORRELATE Y
[1]   XDEV←X-(+/X)÷⍴Y
[2]   YDEV←Y-(+/Y)÷⍴Y
[3]   NUM←+/XDEV×YDEV
[4]   DEN←((+/XDEV*2)×+/YDEV*2)*.5
[5]   NUM÷DEN
      ∇
```

Nice, but does it work?

```
      SUNSPOTS←3 6 8 8 4 7 9 11 13
      PRIME←.115 .125 .14 .145 .13 .16 .165 .17 .17
      SUNSPOTS CORRELATE SUNSPOTS
1
      SUNSPOTS CORRELATE PRIME
0.8820097372
```

It seems to. So there you have it: a tool which demonstrates beyond reasonable doubt that the prime lending rate is related to the sunspot count. No self-respecting economist should be without one.

Note that if we wish to round the calculated correlation coefficient to 2 decimal places, say, using the function *ROUND*, we can't do so. Why not? Because *CORRELATE* does not return an explicit result:

```
      2 ROUND SUNSPOTS CORRELATE PRIME
0.8820097372
RESULT ERROR
      2 ROUND SUNSPOTS CORRELATE PRIME
              ^
```

If we modify the function header so that it reads instead

```
      ∇R←X CORRELATE Y
```

and line 5 so that it reads

```
[5]   R←NUM÷DEN
```

then we can execute an expression such as

```
      2 ROUND SUNSPOTS CORRELATE PRIME
0.88
```

The explicit result returned by *CORRELATE* is passed along as the right argument to the dyadic function *ROUND*.

Those of you who have heard legendary tales of APL "one-liners" might be interested to note that *CORRELATE* could have been written less intelligibly as:

```
      ∇R←X CORRELATE Y
[1]   R←(+/XDEV×YDEV)÷((+/XDEV×XDEV←X-(+/X)÷⍴X)×+/YDEV×YDEV←Y-(+/Y)÷⍴Y)*.5
      ∇
```

Other primitive functions useful in statistical analysis

There are a number of other primitive functions of practical utility to the statistician. Consider two variables, one measuring the height in inches of a sample of fathers, the other the heights of their respective sons:

```
      FATHERS←70 68 69 72 62 73
      SONS←54 55 58 53 57 47
```

If your background includes classical statistics, you probably recognize this data as similar to that studied by Sir Francis Galton in his famous investigations into heredity (work continued later by Karl Pearson). The term "regression" was first used in a statistical sense in Galton's paper of 1886 (although his complete expression was "regression to mediocrity", a description some econometricians no doubt consider appropriate to much of today's literature as well).

Using the defined functions *MEAN*, *VARIANCE*, *RANGE* and the rest, we can subject this data to the same cursory analysis to which we subjected the test scores. A discovery we would soon make is that the total deviation from the mean is the same for both variables, as is the total absolute deviation from the mean:

```
      +/FATHERS-(+/FATHERS)÷⍴FATHERS
0
      +/SONS-(+/SONS)÷⍴SONS
0
      +/|FATHERS-(+/FATHERS)÷⍴FATHERS
16
      +/|SONS-(+/SONS)÷⍴SONS
16
```

The latter equality served as the foundation for one of the important conclusions of Galton's investigations.

One common descriptive statistic we did not calculate for the test scores is the median. To calculate the median height for fathers, we need to find the single middle observation, if there is one, or the arithmetic mean of the two middle observations, after the data has been sorted. We must first, then, sort *FATHERS* in ascending order. This is accomplished in part using the monadic function ⍋ (called **upgrade**), formed by overstriking the shift *H* with the shift *M*. Upgrade applied to a string of numbers *X* does not sort *X*, but rather produces that permutation of the numbers 1 through ⍴*X* which, when used to index *X*, produces *X* sorted in ascending order. For example:

```
      FATHERS
70 68 69 72 62 73
      ⍋FATHERS
5 2 3 1 4 6
```

The result of ⍋*FATHERS* indicates that the fifth element of *FATHERS* is the smallest, the second element the next smallest, and the sixth element the largest. To index (access selected elements only) a variable in APL, simply follow the name of the variable by the index or indices in square brackets:

```
      SONS[1]
54
      FATHERS[2 5 1]
68 62 70
```

The permutation ⍋*FATHERS*, then, can be used as an index to the variable *FATHERS* to produce fathers' heights in ascending order:

```
      FATHERS[⍋FATHERS]
62 68 69 70 72 73
```

To sort *FATHERS* in descending order, we could use the monadic function ⍒, called **downgrade** (shift *G* overstruck with shift *M*):

```
      ⍒FATHERS
6 4 1 3 2 5
      FATHERS[⍒FATHERS]
73 72 70 69 68 62
```

Note that both ⍋ and ⍒ produce what are in effect intermediate results. Why bother with an intermediate result? Why not have a primitive function that produces a sorted string directly? It turns out that the result of ⍋ and ⍒, i.e. the string of grades, or *rank indices*, can be used as the index to a number of variables simultaneously, thereby sorting one variable but maintaining the correspondence between the elements of the sorted variable and the elements of the other variables.

For example, given a variable named *INCOME* containing national income figures for a number of countries, and a variable named *POP* containing population statistics for the same countries, we could calculate per capita income as

```
PERCAP←INCOME÷POP
```

If we then needed to tabulate data for income, population and per capita income, sorted according to descending per capita income, we could calculate and print these variables:

```
DOWNG←⍒PERCAP
PERCAP←PERCAP[DOWNG]
INCOME←INCOME[DOWNG]
POP←POP[DOWNG]
```

The proper correspondence between income, population and per capita income for each country would be maintained.

To return to our median example, we now have the data sorted in ascending order:

```
      ORDFATHERS←FATHERS[⍋FATHERS]
      ORDFATHERS
62 68 69 70 72 73
```

The next step is to calculate the middle observation, if the variable contains an odd number of elements, or the mean of the two middle observations, if the variable contains an even number of elements. The index of the middle element of a string *X* consisting of an odd number of elements is clearly ⌈.5×⍴*X*. If *X* consists of an even number of elements, we need the indices of the two central elements, namely .5×⍴*X* and ⌈.5×1+⍴*X*. These two central indices can be generated by the single expression ⌈.5×0 1+⍴*X*. In our case, *ORDFATHERS* contains an even number of elements:

```
      0 1+⍴ORDFATHERS
6 7
      ⌈.5×0 1+⍴ORDFATHERS
3 4
```

The result 3 4 indicates that we need to extract the third and fourth elements of *ORDFATHERS*, like this:

```
      ORDFATHERS[⌈.5×0 1+⍴ORDFATHERS]
69 70
```

and calculate the arithmetic mean of this result:

```
      (+/ORDFATHERS[⌈.5×0 1+⍴ORDFATHERS])÷2
69.5
```

The median height for fathers in the sample is 69.5 inches. But suppose the number of data points had been odd. How would this have affected our calculations? Let *X* be a string of numbers sorted in ascending order, and let *X* contain an odd number of elements. We then want to extract the single middle element in index position ⌈.5×⍴*X*, and not calculate any mean. If we instead apply the same algorithm as in the even-numbered case, namely:

```
      ⌈.5×0 1+⍴X
```

the resulting indices will be the single middle index repeated twice (consider ⍴*X* to be 7, for example; the resulting indices would be 4 4). The arithmetic mean calculation applied subsequently to this result clearly produces the desired result, albeit in somewhat roundabout fashion.

A monadic function which calculates the median of a string of numbers, then, might look like this:

```
      ∇R←MEDIAN X
[1]   X←X[⍋X]
[2]   R←(+/X[⌈.5×0 1+⍴X])÷2
      ∇

      MEDIAN FATHERS
69.5
      MEDIAN SONS
54.5
```

MEDIAN has the disadvantage that it performs some unnecessary calculations when *X* contains an odd number of elements. On the other hand, it has the advantages that it applies equally well to strings of even and odd numbers of elements, and that whether ⍴*X* is even or odd need not be known beforehand (nor must it be ascertained explicitly by the function). An alternative approach is to decide first whether ⍴*X* is even or odd (according to 2|⍴*X*), and then apply one of two distinct calculations depending on this decision.

As we shall see later, the functions ⍋ and ⍒ are also useful in dealing with heteroscedastic disturbances in econometric problems.

Suppose now that we were to continue our analysis of the variables *FATHERS* and *SONS*, and were able to establish a relationship between the two, as did Karl Pearson, using classical regression methods. (It should be pointed out that Pearson employed a more sensible sample size of 1,078, rather than our paltry 6.) Given this relationship, we could then conduct simulation experiments in an attempt to ascertain how sons' heights were affected by fathers' heights in cases other than those included in the estimation sample. What more appropriate way to generate additional fathers' heights (values of the explanatory variable) than to apply a growth rate? Let's do so. First of all, it would be convenient to reorder the data according to the upgrade of fathers' heights:

```
      ORDFATHERS←FATHERS[⍋FATHERS]
      ORDSONS←SONS[⍋FATHERS]
      ORDFATHERS
62 68 69 70 72 73
      ORDSONS
57 55 58 54 53 47
```

Note that reordering *all* variables in an equation according to the *same* index has no effect upon the estimated coefficients, nor upon the resulting equation statistics; we have not upset the relationship in any way by this reordering. Now that the fathers' heights have been arranged in ascending order, we can guarantee that new "observations" generated via a compound growth rate will be outside the range of the estimation sample. Assume a growth rate of 2%, and that this growth is to be compounded. The first new "observation" is then given by:

```
      1.02×73
74.46
```

or, more generally, by:

```
      1.02×ORDFATHERS[⍴ORDFATHERS]
74.46
```

Our second result depends on this result, since the growth is compounded:

```
      1.02×1.02×ORDFATHERS[⍴ORDFATHERS]
75.9492
```

Our second result is calculated from the previous result. It would be convenient to account for the effects of compounding in a single expression, so that all elements of our desired result could be generated simultaneously. The operator **scan** (denoted by \, shift /) provides such a mechanism. Recall from our earlier discussion of reduction that a monadic operator in APL combines with the primitive function to its left to form a derived function, which then applies to the data to its right. We saw in expressions such as

```
      +/4 7 10 3
24
```

that the primitive function (+) to the left of the operator (/) is in effect inserted between successive pairs of elements in the data string to the right, and the resulting expression executed from right to left. Scan operates in a slightly different fashion. Each element of the result is computed by successively applying, from right to left, the argument function to the data. For example, consider:

```
      +\4 7 10 3
4 11 21 24
```

The second element of the result is the sum of the first two data values, the third element of the result the sum of the first three data values, and so on. The derived function illustrated here is called **plus-scan**, and is equivalent to a cumulative sum. As such, +\ is applicable directly to the generation of cumulative distribution functions, and therefore indirectly to the conduct of such tests as the Kolmogorov-Smirnov goodness of fit. Clearly, +/*X* is equivalent to (+*X*)[⍴*X*] for any numeric string *X*.

The primitive function which combines with scan need not be associative. Consider, for example, **minus-scan**:

```
      -\4 7 10 3
4 ¯3 7 4
      4-7
¯3
      4-7-10
7
      4-7-10-3
4
```

Other primitive functions can combine with scan to form useful derived functions. One example is ⌈\, called **maximum-scan**, which produces the cumulative maximum, or "high water mark", of a series of numbers. It could be used to track record high sales figures over a number of months:

```
      SALES
112 118 132 129 121 135 148 148 136 119 104 118
      ⌈\SALES
112 118 132 132 132 135 148 148 148 148 148 148
```

The derived function we need for our compound growth is ×\, called **times-scan**. ×\ generates a cumulative product:

```
      ×\60 60 24 30
60 3600 86400 2592000
```

Our 2% growth is then compounded as follows:

```
      1.02×1.02
1.0404
      1.02×1.02×1.02
1.061208
      1.02×1.02×1.02×1.02
1.08243216
      ×\1.02 1.02 1.02 1.02
1.02 1.0404 1.061208 1.08243216
      ORDFATHERS[⍴ORDFATHERS]××\1+.02 .02 .02 .02
74.46 75.9492 77.468184 79.01754768
```

These four new values for fathers' heights could then be substituted into our estimated equation, and the resulting estimated sons' heights examined. Recall that the monadic function shape (denoted ⍴) returns the shape of its argument; used dyadically, ⍴ is called **reshape**. The right argument of ⍴ is reshaped into the shape given by the left argument. This means that:

```
      5⍴6
6 6 6 6 6
```

and that our compound growth expression can be written as:

```
      ×\4⍴1.02
```

or as

```
      ×\1+4⍴.02
```

rather than the more explicit

```
      ×\1+.02 .02 .02 .02
```

In general, to apply a *G*-percent growth rate, compounded over *N* "periods" or "observations", to a string of numbers *X*, you can use the expression:

```
X[⍴X]××\1+N⍴G÷100
```

Admittedly, the above example is inane, if not ridiculous. However, the utility of the growth rate expression given above in the context of an econometric model is considerable. Time series are commonly thought to consist of four multiplicative (or sometimes additive) components: trend, cyclical, seasonal, and irregular or random. Economic time series in particular often tend to grow with time; as such, they are often dominated, to varying degrees, by their long-term trend component. A compound growth rate such as ours, when applied to an economic time series, would be incapable of capturing any cyclical, seasonal, or irregular variations. But, the greater the extent to which the series is dominated by its trend component (so that the other effects are minimal), the more faithful a representation of series behaviour a growth rate can provide. Despite its obvious limitations, then, our growth rate expression can be very useful in extending exogenous variables in an econometric model for purposes of simulation.

* * *

There are other functions which facilitate the manipulation of exogenous variables in econometric models. Since the response of a dependent variable to a change in an explanatory variable may not be immediate and complete, one often wants to consider the effects of one or more lags of the explanatory variable. Recall the prime lending rate variable we studied earlier:

```
      PRIME
0.115 0.125 0.14 0.145 0.13 0.16 0.165 0.17 0.17
```

The variable *PRIME* represents a time series PRIME spanning nine periods:

t	1	2	3	4	5	6	7	8	9
$PRIME_t$	.115	.125	.14	.145	.13	.16	.165	.17	.17
$PRIME_{t-1}$	---	.115	.125	.14	145	.13	.16	.165	.17

The time series lagged one period, $PRIME_{t-1}$, spans the eight periods t=2 through t=9. When we calculate $PRIME_{t-1}$, then, we have a choice: we can generate the 8 legitimate lagged values only, thereby losing an observation, or we can generate the 8 legitimate values and pad the front end of the result with a dummy observation, thereby maintaining the time correspondence between the elements of the original series and the elements of the result. Either approach could be preferable, depending on the circumstances. We shall do both.

The monadic function ϕ, called **reverse**, reverses the order of a string of numbers. ϕ is formed by overstriking ○, shift *O*, with |, shift *M*:

```
      ϕPRIME
0.17 0.17 0.165 0.16 0.13 0.145 0.14 0.125 0.115
```

The dyadic version of ϕ is called **rotate**. The data elements in the right argument are rotated according to the integer left argument. If the left argument is positive, data elements are rotated towards the left (and those which were formerly at the left end of the string occupy the vacant positions at the right end). A negative left argument rotates the data towards the right. For example:

```
      3ϕPRIME
0.145 0.13 0.16 0.165 0.17 0.17 0.115 0.125 0.14
      ¯1ϕPRIME
0.17 0.115 0.125 0.14 0.145 0.13 0.16 0.165 0.17
      ¯2ϕPRIME
0.17 0.17 0.115 0.125 0.14 0.145 0.13 0.16 0.165
```

Clearly, ¯1⌽*PRIME* represents PRIME_{t-1} and ¯2⌽*PRIME* represents PRIME_{t-2}. In general, the expression

```
(-K)⌽X
```

produces a time series *X* lagged *K* periods. Note, however, that the first *K* elements of the result are dummy observations, which you might not want.

A glance at the table above suggests that if we don't want a dummy observation on the front of our result, we can obtain PRIME_{t-1} by simply ignoring the last element of PRIME_t. The dyadic APL function ↓ (shift *U*), called **drop**, can be used to drop selected consecutive elements from one end of a string. Its counterpart ↑ (shift *Y*) is called **take**, and is used to select consecutive elements from one end of a string:

```
      3↑PRIME
0.115 0.125 0.14
      ¯1↑PRIME
0.17
      4↓PRIME
0.13 0.16 0.165 0.17 0.17
      ¯2↓PRIME
0.115 0.125 0.14 0.145 0.13 0.16 0.165
```

Elements are taken or dropped from the beginning or the end of the string according to whether the left argument to ↑ or ↓ is positive or negative. Clearly, ¯1↑*PRIME* is equivalent to *PRIME*[⍴*PRIME*], and represents the last element of the series PRIME_t. More importantly, ¯4↓*PRIME* is equivalent to the series PRIME_t lagged four periods (and without the dummy observations at the beginning). In general, the expression

```
(-K)↓X
```

produces a series *X* lagged *K* periods. *K* observations are always lost.

These functions have obvious application to the manipulation of time series variables in econometrics, but they facilitate the calculation of some other common statistical measures as well. One of these is the Durbin-Watson d-statistic, as we'll see later. Another is simple percent change. Consider the prime lending rate again. To calculate the percent change in the prime rate from one sample period to the next, we need the formula:

$$100 \times \frac{(\text{PRIME}_t - \text{PRIME}_{t-1})}{\text{PRIME}_{t-1}}$$

spanning the eight periods t=2 through t=9. We've just seen that ¯1↓*PRIME* is equivalent to $\mathbf{PRIME}_{t-1}$:

```
      ¯1↓PRIME
0.115 0.125 0.14 0.145 0.13 0.16 0.165 0.17
```

It's also evident that 1↓*PRIME* produces $\mathbf{PRIME}_t$ for t=2 through t=9:

```
      1↓PRIME
0.125 0.14 0.145 0.13 0.16 0.165 0.17 0.17
```

The period-to-period percent change of the variable *PRIME*, then, is given by the expression:

```
      100×((1↓PRIME)-¯1↓PRIME)÷¯1↓PRIME
8.695652174 12 3.571428571 ¯10.34482759 23.07692308 3.125 3.03030303 0
```

or, assuming our defined function *ROUND* is still available, by:

```
      2 ROUND 100×((1↓PRIME)-¯1↓PRIME)÷¯1↓PRIME
8.7 12 3.57 ¯10.34 23.08 3.13 3.03 0
```

As we'll see in a moment, rotate and reversal, and take and drop, extend themselves readily to apply to arguments which are arrays.

* * *

Three other primitive functions which merit mention, if not detailed discussion, are *?*, ! and ○. The monadic function *?* (shift *Q*), called **roll**, applied to an argument *X*, produces a pseudo-random number between 1 and *X*. The result is termed *pseudo*-random because, although it satisfies most tests of randomness, it is generated by an explicit and repeatable algorithm. If the argument to roll is a string of numbers, the result is a string of the same length:

```
      ?7 20 50 100
1 16 23 54
      ?7 20 50 100
2 1 34 68
```

The first element of the result is a pseudo-random number between 1 and 7, the third a pseudo-random number between 1 and 50, and so on.

Used dyadically, ? is called **deal**, and allows you to generate any number (as specified by the left argument) of pseudo-random numbers between 1 and the limit or limits given in the right argument. For example,

```
      4?100
14 76 46 54
      4?100
22 5 68 94
```

generates four pseudo-random numbers between 1 and 100.

Roll and deal are very useful in the performance of Monte Carlo simulation experiments. An expression such as 4?100 above represents sampling without replacement, while an expression such as

```
      ?4⍴100
39 52 84 4
```

is an example of sampling with replacement (since the elements of the latter result need not be distinct). In particular,

```
      ?6 6
6 3
```

simulates a single roll of a pair of dice, and

```
      ¯1+?10⍴2
0 1 1 1 1 0 1 1 0 0
```

represents a succession of ten Bernoulli trials.

The monadic function ! is formally called **factorial**, although the symbol itself is commonly known as **shriek**. ! is formed by overstriking ' (shift *K*) with . (period), and produces the factorial of each element in its argument:

```
      !4
24
      ×/1 2 3 4
24
      !1 2 3 4 5
1 2 6 24 120
      ×\1 2 3 4 5
1 2 6 24 120
```

Note that the statistical recurrence relation

$$\Gamma(Z + 1) = Z!$$

has the consequence that the APL expression !Z produces the value of the gamma function of Z+1. Such values can be used directly in the generation of other probability distribution functions, including the beta and Student's t.

Used dyadically, ! is called **binomial coefficients**. It produces the number of distinct ways in which the number of items specified in the left argument can be selected from a set containing the number of items specified in the right argument. This is the familiar concept of **combinations** in statistics:

```
      2!6
15
      3!6
20
```

3!6 produces the number of distinct ways in which three items can be selected from a set of six. A common application of dyadic ! is the determination of coefficients of the terms of the binomial expansion. As an illustration of this, consider the following sequence:

```
      0!0
1
      0 1!1
1 1
      0 1 2!2
1 2 1
      0 1 2 3!3
1 3 3 1
      0 1 2 3 4!4
1 4 6 4 1
      0 1 2 3 4 5!5
1 5 10 10 5 1
      0 1 2 3 4 5 6!6
1 6 15 20 15 6 1
```

Clearly, each successive result constitutes a new row in Pascal's triangle. ! is directly applicable to the calculation of probabilities based on the binomial distribution; examples commonly found in introductory texts include calculating the probability of obtaining a certain number of heads or tails in a succession of coin tosses, certain combinations in a series of dice rolls, or certain numbers of children of one sex in a sample of families.

Finally, the monadic function ○ (shift *O*), is formally named **pi times**, but the symbol itself is commonly known as **circle**. The result produced by pi times is the constant π times the one or more numbers in its argument:

```
      ○1 2 .5
3.141592654 6.283185307 1.570796327
```

Dyadic use of ○ encompasses the **family of circular functions**, including trigonometric functions, Pythagorean functions, and hyperbolic functions. The particular circular function to be applied to the right argument is designated by the left argument. The two most common circular functions in practice are 1○*X* and 2○*X*, representing the **sine** and **cosine** of *X* respectively:

```
      ○¯1 ¯.5 0 .5 1
¯3.141592654 ¯1.570796327 0 1.570796327 3.141592654
      1○○¯1 ¯.5 0 .5 1
0 ¯1 0 1 0
      2○○¯1 ¯.5 0 .5 1
¯1 0 1 0 ¯1
```

The last two lines above evaluate the sine and cosine functions respectively for each of the values. $-\pi, \frac{-\pi}{2}, 0, \frac{\pi}{2}, \pi.$

The statistical utility of the circular functions lies primarily in their ability to capture cyclical behaviour in a time series, a series whose cyclical component might follow a sinusoidal pattern, for example. Another obvious application would be the implementation of Fourier transforms, and the general study of a series within the frequency domain (spectral analysis), rather than within the time domain (time series analysis).

Arrays

The statistics and econometrics textbooks on which we've been nurtured tend to employ one of two basic mathematical notations. The first is the familiar element-by-element notation, in which each individual observation is represented directly:

$$y_i = \beta_0 + \sum_{j=1}^{k} \beta_j x_{ij} + \varepsilon_i \qquad i = 1, \ldots, n$$

Note that this is conceptually a sort of explicit matrix notation, in that explicit reference is made to a row subscript i and a column subscript j into a data matrix consisting of n rows and k columns. This notation carries with it the suggestion that any required calculations must be performed iteratively, i.e. to calculate the n values of Y, we must loop over the k different values of the subscript j for each and every one of the n values of the subscript i.

The second basic notation employed is commonly known as matrix notation. Each bold-face character represents an entire matrix of data:

$$Y = X\beta + \varepsilon$$

Note that this notation bears the implicit suggestion that the matrix **X** can be multiplied by the matrix β in a single step, by means of a single operation. Perhaps we do not have to make explicit reference, via subscripts and iteration, to each individual element of the matrices **X** and β .

Most languages other than APL impose upon us the element-by-element notation given first above. That is, we are forced by other languages to perceive a problem as it pertains to an individual data observation, and to perform all necessary calculations in the same manner. Such calculations invariably involve tedious iterative algorithms, and can make the programming experience a stultifying one. Problems can be approached and solved in this manner using APL as well. However, they need not be. APL allows us to perceive a problem as it applies to an entire array of data (such as a matrix, for example), and to perform calculations on one or more arrays using a single expression, rather than performing these same calculations element-by-element. Primitive APL functions, some of which we've already seen, provide us with a means of "translating" a textbook formula in matrix notation into a single expression. Using these functions, we can represent such a complicated relationship as:

$$b^* = (X'X)^{-1}X'y + (X'X)^{-1}R'[R(X'X)^{-1}R']^{-1}[r - R(X'X)^{-1}X'y]$$

as an expression which can be read and interpreted in much the same way as can the original. APL's ability to manipulate arrays is probably its most useful and pertinent feature to a statistician or econometrician.

* * *

To this point, every primitive function we've seen has been applied to a single number or to a string of numbers. But in econometrics, in particular, it is convenient to work with matrices. A matrix is an array of two dimensions, or **axes**: rows and columns. How do we define a matrix? Consider the data in the following table:

Consumer Price Index (1975 = 100)				
	1977	1978	1979	1980
U.S.A.	112.7	121.2	134.9	153.1
Canada	116.1	126.5	138.1	152.1
U.K.	135.0	146.2	165.8	195.6

To define this as an APL matrix, we need to enter twelve numbers in such a way that they fill out an array consisting of three rows and four columns. Recall that the dyadic function ρ reshapes its right argument according to the shape specified in its left argument, thus:

```
      4ρ8
8 8 8 8
      2 4ρ1 2 3 4 5 6 7 8
1 2 3 4
5 6 7 8
      2 3ρ6 8 9 4
6 8 9
4 6 8
```

In the last expression above, the left argument 2 3 indicates that the result is to have two axes, the first of length 2 and the second of length 3. Note, however, that 2x3=6 numbers are to constitute the result, while only four numbers have been provided in the right argument. ρ always repeats cyclically the data to its right until the shape specified to its left has been filled. In the case of a matrix, as we have, the first row is filled across all columns, then the second row across all columns. In general, the last axis is filled first, from left to right, then the second-last axis, and so on until the complete shape requirement specified by ρ's left argument has been fulfilled. For example:

```
      THREED←2 3 4ρ1 2 3 4 5 6 7 8 9 10
      THREED
 1  2  3  4
 5  6  7  8
 9 10  1  2

 3  4  5  6
 7  8  9 10
 1  2  3  4
```

THREED is an array of three axes. It consists of two "planes", each of which is a matrix of three rows and four columns. Observe the sequence in which the numbers 1 through 10 are recycled through the last axis (columns) first. An APL array can have as many as 63 or more axes, depending on the implementation; we will restrict our discussion to matrices.

Our consumer price index matrix, then, is constructed as follows:

```
      CPI←3 4ρ112.7 121.2 134.9 153.1 116.1 126.5 138.1 152.1 135
                    146.2 165.8 195.6
      CPI
112.7 121.2 134.9 153.1
116.1 126.5 138.1 152.1
135   146.2 165.8 195.6
      ρCPI
3 4
```

Note that the shape of *CPI* consists of two numbers. The shape of the shape of a variable (ρρ*X*) in APL is called the **rank** of that variable. A single number in APL is called a **scalar**. Its shape is empty; its rank is zero. A string of numbers (such as *SCORES* or *PRIME*) is called a **vector**. Its rank is one, since its shape is a single number. And, as we've seen with *CPI*, an array of two axes is called a **matrix**, and has rank two. To illustrate:

```
      ⍴5
```
(the shape of a scalar is an empty vector)
```
      ⍴⍴5
0
      ⍴1 2 3 4
4
      ⍴⍴1 2 3 4
1
      ⍴7 20⍴140?200
7 20
      ⍴⍴7 20⍴140?200
2
```

Arrays consisting of more than two axes have no special names associated with them; they are referred to simply as arrays of rank n.

We have already seen that primitive APL functions apply equally well to both scalars and vectors. This is true of matrices and arrays of higher rank as well:

```
      CPI÷100
1.127 1.212 1.349 1.531
1.161 1.265 1.381 1.521
1.35  1.462 1.658 1.956
```

The dyadic function ÷ above could just as easily have been any of +, -, ×, ⌈, ⌊ or a number of others. For example, if we wished to proceed under the illusory conviction that the CPI should never exceed 150, we could examine

```
      150⌊CPI
112.7 121.2 134.9 150
116.1 126.5 138.1 150
135   146.2 150   150
```

The expressions *CPI*÷100 and 150⌊*CPI* illustrate the use of functions taking a matrix and a scalar as arguments. These dyadic functions can take a matrix and a matrix as arguments as well:

```
      2 2⍴1 2 3 4
1 2
3 4
      2 2⍴0 3 4 3
0 3
4 3
      (2 2⍴1 2 3 4)-2 2⍴0 3 4 3
 1 ¯1
¯1  1
      3 3⍴4 8 12 16
 4  8 12
16  4  8
12 16  4
      3 3⍴1 0 0 0
1 0 0
0 1 0
0 0 1
      (3 3⍴4 8 12 16)×3 3⍴1 0 0 0
4 0 0
0 4 0
0 0 4
```

In these examples the dyadic functions - and × are applied element-by-element to their two arguments, producing a result of the same shape as the shape of each argument. Such operations between two matrices are possible by means of a single function, without explicit reference to each pair of individual elements, as long as the two matrices are of the same shape.

Consider again the matrix *CPI*, and suppose we wish to calculate the average index over the four years for each of the three countries. We first need to sum each row of *CPI*, no doubt employing the derived function +/. But how is APL to interpret the expression +/*CPI*? Should +/*CPI* sum along the three rows, or down the four columns? Or should it sum all the elements of the matrix, without regard to row and column index? Note that if +/*CPI* did sum all elements of the matrix, this would mean that +/ could not be used to produce such results as row and column totals. In fact, +/ sums an array along whichever axis is specified. The axis is indicated by a number enclosed in brackets (so that axis selection is similar in appearance to an index into a vector). Obviously, the axis number must be between one and the rank of the array. For example, the expression +/[1] *CPI* indicates that the derived function +/ is to apply along the first axis (rows) of the matrix *CPI*:

```
      +/[1] CPI
363.8 393.9 438.8 500.8
```

Similarly, +/[2] *CPI* indicates that +/ is to apply along the second axis, i.e. across columns:

```
      +/[2] CPI
521.9 532.8 642.6
```

Clearly, +/[1] *CPI* produces column totals, while +/[2] *CPI* produces row totals. The average index for each of our three countries over the four years is therefore:

```
      (+/[2] CPI)÷4
130.475 133.2 160.65
```

or, more generally:

```
      (+/[2] CPI)÷¯1↑⍴CPI
130.475 133.2 160.65
```

Similarly, the average CPI of all three countries in each of the four years is:

```
      (+/[1] CPI)÷1↑⍴CPI
121.266667 131.3 146.266667 166.9333333
```

Note that plus-reduction applied along either axis of *CPI* reduces a matrix to a vector, so that its result has rank one less than the rank of its argument.

By convention, if the reduction is to apply along the last axis, then the axis need not be specified. That is,

```
      +/ARRAY
```

is equivalent to

```
      +/[⍴⍴ARRAY] ARRAY
```

for any non-scalar array *ARRAY*. This means that the sum of all twelve elements of *CPI* can be obtained using the expression

```
      +/+/CPI
1697.3
```

Here, summation is applied along the last axis first (columns), and the resulting three subtotals are summed. Alternatively:

```
      +/+/[1] CPI
1697.3
```

The average index over all three countries and all four years is then:

```
      (+/+/CPI)÷12
141.4416667
```

or, more generally,

```
      (+/+/CPI)÷×/⍴CPI
141.4416667
```

Rather than applying reduction twice in succession, thereby reducing the rank of *CPI* by two (from a matrix to a scalar), we could instead transform *CPI* into a vector of twelve numbers, and sum that vector. The monadic function , called **ravel**, converts its argument into a vector:

```
      ,CPI
112.7 121.2 134.9 153.1 116.1 126.5 138.1 152.1 135 146.2 165.8 195.6
      (×/⍴CPI)⍴CPI
112.7 121.2 134.9 153.1 116.1 126.5 138.1 152.1 135 146.2 165.8 195.6
```

The sequence in which elements of an array are "unravelled" into a string is similar to that in which the elements of a vector are selected to fill an array when using reshape. That is, in the case of a matrix, all columns of the first row are unravelled first, followed by all columns of the second row, and so on. The result of the expression ,*CPI* is therefore a string of numbers identical to that originally supplied as the right argument to reshape when we defined the matrix *CPI* (i.e. following *CPI*←3 4⍴). Clearly, +/,*CPI* is equivalent to +/+/*CPI*:

```
      +/,CPI
1697.3
```

so that the average of all twelve indices could also be calculated as:

```
      (+/,CPI)÷⍴,CPI
141.4416667
```

Ravel's dyadic counterpart, called **catenate**, serves a useful purpose as well. Catenate forms a new array by joining together the two arrays that are its arguments:

```
      FATHERS
70 68 69 72 62 73
      SONS
54 55 58 53 57 47
      FATHERS,SONS,50
70 68 69 72 62 73 54 55 58 53 57 47 50
```

The two argument arrays need not be of the same rank: a matrix can be catenated to another matrix, or a vector to a matrix, or a matrix to a scalar. The only restriction applied to catenate is that its arguments must be of conformable shape; more on conformability in a moment. Catenate is the vehicle by which we append a new row or column to a matrix. Again, we have to specify the axis along which the function (catenation) is to apply:

```
      CPI←CPI,169 171 218.8
      CPI
112.7 121.2 134.9 153.1 169
116.1 126.5 138.1 152.1 171
135   146.2 165.8 195.6 218.8
```

Omission of the axis specification indicates that , is to apply over the last axis (i.e. columns) of the matrix *CPI*. The three new numbers in the fifth column represent the price indices for 1981. We can append a new row, representing the CPI for Australia for the five years 1977 through 1981, in a similar manner:

```
      CPI←CPI,[1] 113.2 117.3 121.6 129.3 138.1
      CPI
112.7 121.2 134.9 153.1 169
116.1 126.5 138.1 152.1 171
135   146.2 165.8 195.6 218.8
113.2 117.3 121.6 129.3 138.1
```

We must specify that the catenation is to apply along the first axis. Note that the catenation is successful only because the shape of the right argument (5) is conformable with that of the left argument (3×5): a new row of *CPI* must span five columns. A scalar can be catenated as well, in which case its single value is repeated over the entire row (or column). However, a vector of length other than five, given that *CPI* has five columns, is not acceptable as a right argument to , . In general, to a matrix of shape n×k the following can be catenated, along the first axis:

- a scalar
- a vector of length k
- a matrix with k columns

or, along the second axis:

- a scalar
- a vector of length n
- a matrix with n rows

Catenation is a convenient means of building a matrix of explanatory variables in a regression model, and of appending new explanatory variables to that matrix.

* * *

We saw earlier how the functions rotate and reversal (dyadic and monadic ⌽, respectively) can be used to modify the sequence of a vector. These functions apply to matrices and arrays of greater rank as well. Once again, the axis along which the function is to apply can be specified. For example, the expression ⌽[1] *CPI* reverses the order of the rows of the matrix *CPI* (turning *CPI* upside down), while ⌽*CPI* reverses the order of its columns (producing a mirror image). Similarly, ¯1⌽*CPI* rotates the columns of *CPI* one position towards the right, so that columns other than the first contain the price indices of the previous year (lagged values):

```
      ¯1⌽CPI
169   112.7 121.2 134.9 153.1
171   116.1 126.5 138.1 152.1
218.8 135   146.2 165.8 195.6
138.1 113.2 117.3 121.6 129.3
```

Another monadic function which affects the structure of an array is ⍉, called **transpose**. ⍉ is formed by overstriking ○ (shift *O*) with \ (shift /). Transpose reverses the sequence of the axes of its argument. When the argument is a matrix, transpose converts the rows into columns, and the columns into rows, so that the resulting matrix is the original turned on its side:

```
      ⍉CPI
112.7 116.1 135   113.2
121.2 126.5 146.2 117.3
134.9 138.1 165.8 121.6
153.1 152.1 195.6 129.3
169   171   218.8 138.1
      ⍴CPI
4 5
      ⍴⍉CPI
5 4
```

In general, for any array *X*, if *Z*←⍉*X*, then ⍴*Z* is equivalent to ⌽⍴*X*. Clearly, transpose applied to a matrix argument is one of the fundamental tools we require to translate textbook matrix notation into executable APL expressions.

Application of the dyadic functions take and drop also extends readily to arrays of rank greater than one. Recall that the left argument to ↑ and ↓ indicates by its sign from which end of a vector elements are to be selected, and by its magnitude how many elements are to be selected. But if the right argument to ↑ or ↓ (the data array) has rank greater than one, say *k*, then how is the (single number in the) left argument to be interpreted? The answer is that the left argument must consist not of a single number, but of *k* numbers referring to each of the *k* axes of the data array.

Consider, for example, the case in which the data array is a matrix. The first of the two numbers in the left argument indicates how many rows (first axis) are to be selected, the second how many columns (second axis). First or last rows and columns are selected according to whether the left argument numbers are positive or negative respectively:

```
      2 ¯3↑CPI
134.9 153.1 169
138.1 152.1 171
      0 ¯2↓CPI
112.7 121.2 134.9
116.1 126.5 138.1
135   146.2 165.8
113.2 117.3 121.6
      ρ2 ¯3↑CPI
2 3
      ρ0 ¯2↓CPI
4 3
```

The first expression above selects the first two rows and the last three columns of the matrix *CPI*, producing a matrix of shape 2×3. The second expression selects everything but the first zero rows and the last two columns, producing a matrix of shape 4×3. No rows have been dropped. Note that the shape of this result is the shape of the right argument less the absolute value of the left argument:

```
      (ρCPI)-|0 ¯2
4 3
```

Take and drop allow us to "lag" entire matrices. As an illustration, `0 1↓CPI` could be considered the "current period" index for the years 1978 through 1981, and `0 ¯1↓CPI` the "previous period" index for the same years. The year-to-year percent change in *CPI*, then, over all years for all four countries simultaneously, can be calculated using the single expression:

```
      100×((0 1↓CPI)-0 ¯1↓CPI)÷0 ¯1↓CPI
7.542147294 11.30363036  13.49147517  10.38536904
8.957795004  9.169960474 10.13758146  12.4260355
8.296296296 13.40629275  17.973462    11.8609407
3.621908127  3.665814152  6.332236842  6.805877804
```

or, again applying the defined function *ROUND*:

```
      1 ROUND 100×((0 1↓CPI)-0 ¯1↓CPI)÷0 ¯1↓CPI
7.5 11.3 13.5 10.4
9    9.2 10.1 12.4
8.3 13.4 18   11.9
3.6  3.7  6.3  6.8
```

* * *

To this point we've seen how to create arrays using reshape, and to alter their structure using a variety of functions. However, APL features other array manipulation capabilities more powerful than those we've seen so far. Because of the abundance of matrix notation employed in econometric exposition, we find practical application of these capabilities in the field of econometrics.

Recall first the fundamental concept of a Cartesian product. The **Cartesian product** of two sets A and B is the set consisting of all possible ordered pairs formed when the first element is selected from set A and the second from set B. The concept of a Cartesian product is represented in APL by the derived function **outer product**. The syntax of this derived function is:

```
X∘.gY
```

where g is any one of a number of dyadic primitive functions, many of which we've discussed. The symbol ∘ (shift *J*) is called **null**. The symbol . , called **dot**, is another example of an APL operator; more on that in a moment. *X*∘.g*Y* is called the **g outer product** of *X* and *Y*. A simple example is the **times outer product**, which produces a common multiplication table:

```
      10 20 30∘.×1 2 3 4 5
 10   20   30   40   50
 20   40   60   80  100
 30   60   90  120  150
```

The first element of the left argument, 10, is multiplied by every element of the right argument, and these five simple products constitute the first row of the result. The second and third rows of the result are formed in a similar fashion. In producing the result, the dyadic function × is evaluated for every possible pairing of an element of the left argument with an element of the right argument. The number of rows in the result is therefore the number of elements in the left argument (vector), and the number of columns in the result the number of elements in the right argument (vector).

Outer product must always be used dyadically. The two arguments need not be vectors, however; they can be arrays of any rank. In general, the shape of the result produced by *X*∘.g*Y* is (ρ*X*),ρ*Y*, the rank of the result being therefore the sum of the ranks of *X* and *Y*. Clearly, use of outer product with two vector arguments is a means of constructing a matrix without using ρ.

The dyadic function g following the dot can be any of +, -, ×, ÷, *, ⊛, ⌈, ⌊, |, !, or ○, among others. For example, the derived function ∘.+ generates a table of indices:

```
      0 1 2 3∘.+1 2 3 4 5
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

On the surface, this result seems less than spectacularly useful. The progression of its columns, however, makes the table a valuable tool for calculating moving averages, as we'll see later.

Dyadic primitive functions which can be used in the derivation of outer products also include the class of relational functions: =, ≠, >, <, ≥, and ≤. Each of these functions proposes an equality or inequality between two conformable arrays (of any rank). The result of each function is an array in which an element is 1 whenever the proposed relationship between the corresponding elements of the two argument arrays is true, and 0 if it is false. For example:

```
      2=1 2 3
0 1 0
      3 6 7 4>5 6 4 1
0 0 1 1
      0≠3|3 3⍴1 2 3 4 5 6 7 8 9
1 1 0
1 1 0
1 1 0
```

When employed in outer products, relational functions are helpful in generating matrices often required in statistical calculations. One example of this utility is the construction of an identity matrix:

```
      1 2 3 4∘.=1 2 3 4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The derived function ∘.= has other useful applications. Given a vector *RESPONSES* containing coded questionnaire responses, and another vector *POSSIBLES* containing a list of all possible responses, the expression

```
POSSIBLES∘.=RESPONSES
```

produces an incidence map showing where the various possible responses were actually received, and

```
+/POSSIBLES∘.=RESPONSES
```

counts the number of times that each possible response was received.

Note that the construction of an identity matrix need not employ an outer product. An identity matrix of order n could instead be generated, using reshape, according to the expression $(N,N)\rho 1,N\rho 0$. For example,

```
      4 4ρ1,4ρ0
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The same cannot be said of such similar matrices as triangular matrices. That is, there exists no simple and convenient expression, using reshape, to construct a triangular matrix. Such a matrix can, however, be constructed easily using a relational function in an outer product:

```
      V∘.≥V←1 2 3
1 0 0
1 1 0
1 1 1
      V∘.≤V←1 2 3 4 5
1 1 1 1 1
0 1 1 1 1
0 0 1 1 1
0 0 0 1 1
0 0 0 0 1
      M←4 4ρ7 26 14 8 13 15 3 19 22 24 20 18 6 8 10 12
      M
 7 26 14  8
13 15  3 19
22 24 20 18
 6  8 10 12
      MxV∘.≤V←1 2 3 4
 7 26 14  8
 0 15  3 19
 0  0 20 18
 0  0  0 12
```

Triangular matrices such as that of the last example shown above are common in statistical computation. They are, for instance, fundamental to the Newton-Raphson algorithm employed by Box and Jenkins in the preliminary estimation of the moving average parameters of an ARIMA model.

The dot symbol employed in an outer product represents an APL operator. The formal name of the operator is **product**. Product differs from such other operators as reduction and scan in that it is dyadic. It must be used with two arguments. (Note that the derived function outer product that results from the application of the operator is also dyadic.) Each argument must be a primitive function. But in an outer product such as ∘.+, the only function that need be applied is + . Why bother, then, with a left argument? Why not have product operate monadically? The existence of a left argument means that we can apply that function to the results of the function given in the right argument. The null symbol as the left argument of an outer product indicates that there is no second function, and that only the function in the right argument is to apply between the two data arrays.

The derived function resulting from the application of the product operator to two primitive functions is called **inner product**. Any of the primitive functions which is valid in an outer product is valid as either argument in an inner product. Common examples are +.× , called **plus-dot-times inner product**, and -.÷ , called **minus-dot-divide inner product**. We noted that the derived function outer product is always dyadic; the same is true of most, though not all, APL implementations of the derived function inner product.

An inner product operates in much the same way as does an outer product, with one basic difference. The outer product *X*∘.g*Y*, in forming a Cartesian product, pairs each individual element (scalar) of the left argument *X* with each individual element (scalar) of the right argument *Y*. By contrast, the inner product *X*f.g*Y* pairs each vector along the **last** axis of *X* with each vector along the **first** axis of *Y*. The dyadic function g is first applied between corresponding elements of each pair of vectors, and the derived function f/ then applied to the array that results. A couple of examples involving vector arguments should help illustrate the concept of an inner product:

```
      2 3 4+.×1 3 5
31
      +/2 3 4×1 3 5
31
```

These two expressions are equivalent. In each, the dyadic function × is first applied between successive elements of the two arguments, and the derived function +/ is then applied to the three-element vector that results. The two expressions which follow are also equivalent; each counts the number of corresponding elements in two vectors that match:

```
      14 16 17 19 21+.=12 16 17 18 21
3
      +/14 16 17 19 21=12 16 17 18 21
3
```

An inner product can also be used to calculate a series approximation to the exponential base *e*, namely:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} \cdots$$

To approximate *e* using a series of n terms, we need to generate factorials of the numbers 1 through n, calculate their reciprocals, and sum the reciprocals. The monadic function ⍳ (shift *I*), called **index generator**, applied to a scalar argument *N*, generates the integers 1 through *N*:

```
      ⍳5
1 2 3 4 5
```

Given n, the number of terms to be used in the series expansion, we can then approximate *e* using the expression:

```
      N←20
      1+1+.÷!⍳N
2.71828128
      1++/÷!⍳N
2.71828128
      *1
2.71828128
```

In this application the left argument to the derived function +.÷ is a scalar (1). It is divided by the factorials of the first twenty integers, and the twenty resulting reciprocals are summed. In a similar manner, the series approximation to *e* , namely:

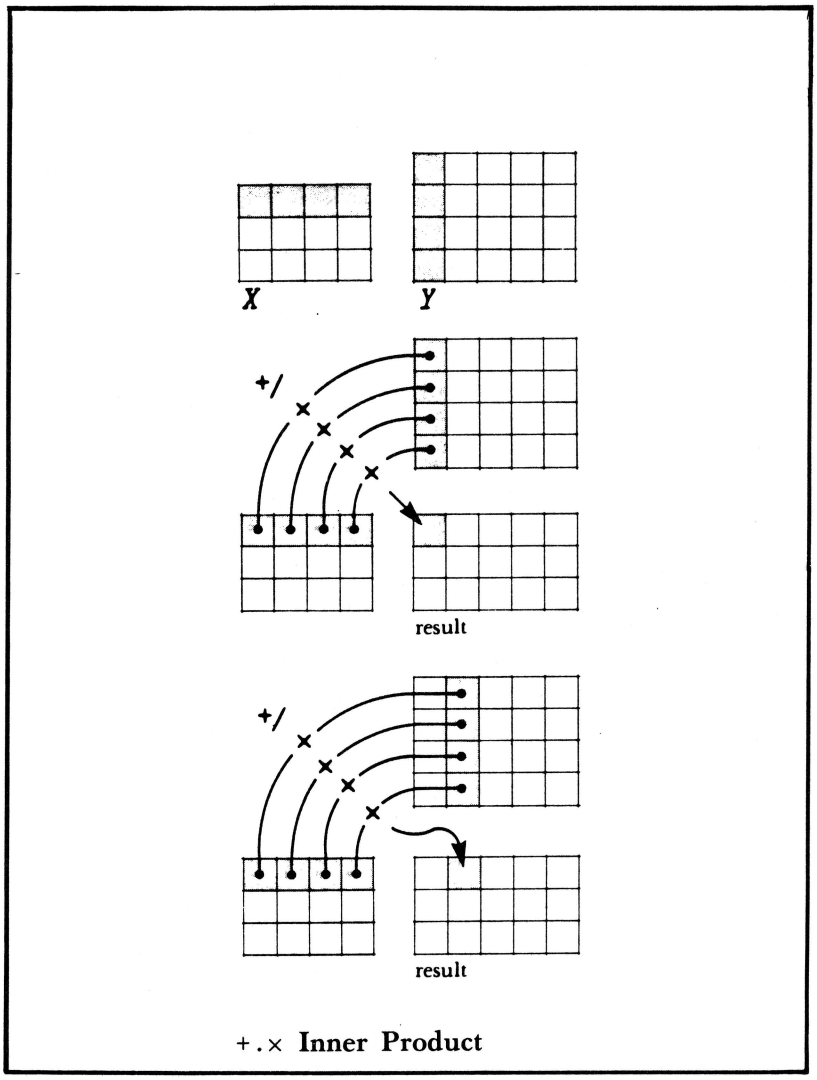
$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

can be computed (given *N*) according to:

```
      1+(X*⍳N)+.÷!⍳N
```

Series approximations to such trigonometric functions as sine and cosine (approximations which involve an alternating sum) can also be generated using the inner product -.÷ .

These illustrations demonstrate the operation of inner products on scalar and vector arguments. But how does inner product apply to arrays of rank greater than one (matrices, for example)? Consider the derived function +.× again, but suppose that instead of being vectors, the arguments are a 3×4 matrix *X* and a 4×5 matrix *Y*:



+.× Inner Product

Again, the dyadic function × applies between pairs of elements of *X* and *Y*, and the function +/ then applies over the resulting products. Recall that the inner product pairs each vector along the *last* axis of *X* with each vector along the *first* axis of *Y*. In our case, then, +.× pairs each *row* of *X* (along the columns of *X*) with each column of *Y* (along the rows of *Y*). The four elements in the first row of *X* are therefore multiplied by the corresponding four elements in the first column of *Y*; the four resulting products are summed, and their sum is the element in row 1 and column 1 of the result. The element in row 1 and column 2 of the result is generated from the vectors in row 1 of *X* and column 2 of *Y*, and so on. In general, the element in row i and column j of the result matrix is calculated using the vectors from row i of the left argument matrix and column j of the right argument matrix. In our case, there are 3×5=15 possible pairings of the rows of *X* with the columns of *Y*; the result matrix therefore contains 15 elements, shaped into three rows and five columns. In general, the shape of the result of an inner product is:

```
ρXf.gY  ↔  (¯1↓ρX),¯1↓ρY
```

Note that APL would not have known how to pair elements of *X* and *Y* had the number of columns in *X* (four) not matched the number of rows in *Y*. The arguments to an inner product must always be conformable in that the last axis of the left argument must be the same length as the first axis of the right argument.

You no doubt recognize that *X*+.×*Y* performs the matrix multiplication of two matrices *X* and *Y*, i.e. *Z*←*X*+.×*Y* is equivalent to the familiar matrix equation:

$$\mathbf{Z} = \mathbf{XY}$$

Individual elements of the matrix **Z** are calculated according to:

$$z_{ij} = \sum_{k=1}^{m} x_{ik}\ y_{kj}$$

where m is the number of columns in **X**. Inner products in general, and the derived function +.× in particular, are therefore extremely useful tools in econometric practice. For example, a matrix fundamental to econometric computation is the moment matrix, or sum of squares and cross-products (SSCP) matrix, usually denoted in the literature by **X'X** or $\mathbf{X^TX}$. This matrix is computed very simply in APL according to the expression:

```
(⍉X)+.×X
```

* * *

We have now seen how to construct matrices, using reshape and outer products, how to modify their structure using such functions as catenate, rotate and transpose, and how to perform computations between them using both primitive and derived functions (such as inner product). One final APL implement completes our econometric toolbox. It is probably the primitive function of the greatest labour-saving utility to the practicing econometrician.

The monadic function ⌹ is called **matrix inverse**. The symbol ⌹, called **domino**, is formed by overstriking ⌷ (shift *L*) with ÷ (shift ×). Matrix inverse applied to a matrix *X* produces the inverse of *X*, that is, the matrix which when matrix-multiplied by *X* produces the identity matrix. (The identity matrix is the matrix equivalent of unity.) Recall that in matrix notation, the inverse of a matrix **X** is a matrix $\mathbf{X}^{-1}$ such that:

$$\mathbf{I} = \mathbf{X}^{-1}\mathbf{X}$$

Since the algebraic concept of matrix multiplication is represented in APL by the inner product +.×, this equivalence is written in APL as:

```
I ↔ (⌹X)+.×X
```

For example:

```
      M←4 4⍴1 2 4 2 7 4 5 6 7 2 2 2 2 2 4 1
      M
1 2 4 2
7 4 5 6
7 2 2 2
2 2 4 1
      ⌹M
  1.25   ¯0.5    0.75   ¯1
¯10.875   3.75  ¯4.625   8.5
  4.25   ¯1.5    1.75   ¯3
  2.25   ¯0.5    0.75   ¯2
      (⌹M)+.×M
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
      M+.×⌹M
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

An argument *X* to ⌹ need not be square. However, the columns of *X* must be linearly independent. Matrix inverse applies as well to an over-identified matrix, i.e. to a matrix with more rows than columns, but for purposes of our discussion we need concern ourselves only with matrix arguments to ⌹ which are square.

The labour-saving consequences of the capabilities of matrix inverse are substantial. Note that ⌹ is a primitive function in APL, as is + or ×, and not a subroutine or auxiliary program that must be invoked by some other program. The availability of ⌹ means that such everyday econometric computations as the inversion of the moment matrix, i.e.

$$(X'X)^{-1}$$

can be accomplished using a single APL expression:

```
⌹(⍉X)+.×X
```

More impressively, consider the general linear model estimated under the assumptions of ordinary least squares (OLS):

$$Y = X\hat{\beta} + \varepsilon$$

and recall that the estimated coefficients which minimize the sum of squared residuals $\varepsilon'\varepsilon$ are given by:

$$\hat{\beta} = (X'X)^{-1}X'Y$$

The estimated coefficients of an OLS regression, then, can be calculated according to the single APL expression:

```
BETA←(⌹(⍉X)+.×X)+.×(⍉X)+.×Y
```

APL has provided us with all the tools we need to perform a detailed econometric analysis, as we'll see in the brief examples which follow.

APL in applied econometrics

Finally, consider yourself a user of an APL timesharing system, a user interested in performing econometric analysis. You know that you have available all the tools required to write your own regression program (to perform ordinary least squares, at least), and you may wish to do so. This is unnecessary, however, since the various APL suppliers offer public software (often referred to as "canned packages"). Suppose for the time being that you prefer to use a supplied regression utility, rather than taking the time (not much!) to write your own.

Canned regression programs in APL typically have names like *REG* or *REGRESS*. Rather than using such a name, and thereby making inadvertent reference to the software of a particular APL vendor, we'll use something different: suppose that the ordinary least squares program available is named *PEGGYLEE*. Since the assumptions of OLS are implicit in the program, the only information *PEGGYLEE* requires is the data. That is, *PEGGYLEE* could have syntax

```
PEGGYLEE X
```

where the single argument *X* is a matrix whose rows (or columns) represent variables. But some means, or convention, would then be required by which we could indicate which of the variables is to be the dependent variable. *PEGGYLEE* is probably more useful, then, if it is dyadic:

```
Y PEGGYLEE X
```

The left argument *Y* is a vector representing the dependent variable, and the right argument *X* a matrix whose columns contain the independent variables.

In terms of the general linear model:

$$Y = X\beta + u$$

or,

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & & & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}$$

the matrices **Y** and **X** constitute the two arguments to *PEGGYLEE*. The column of 1s in **X** represents the constant term β_0, and should not be included if the fitted regression curve is to pass through the origin.

Clearly, then, the only APL required to execute *PEGGYLEE* is that needed to construct the data matrix **X**. This is hardly a restriction; in fact, it is a convenience which allows us to specify the form, as well as the content, of the equation being estimated. Implicit nonlinearities are introduced by applying appropriate transformations when constructing **X**, as we'll see in a moment.

As a simple illustration of the application of *PEGGYLEE*, recall the variables *SUNSPOTS* and *PRIME*:

```
      SUNSPOTS
3 6 8 8 4 7 9 11 13
      PRIME
0.115 0.125 0.14 0.145 0.13 0.16 0.165 0.17 0.17
```

We suspected that there exists a relationship between the two. Let us now attempt to quantify that relationship:

```
      SUNSPOTS∘.*0 1
 1  3
 1  6
 1  8
 1  8
 1  4
 1  7
 1  9
 1 11
 1 13
       PRIME PEGGYLEE SUNSPOTS∘.*0 1
0.1025833333 0.00575
```

An outer product is used to construct the matrix **X** of explanatory variables. The result of the outer product contains ρ*SUNSPOTS* (nine) rows and ρ0 1 (two) columns. The first column contains the nine values of *SUNSPOTS*⋆0, and the second the nine values of *SUNSPOTS*⋆1. The two printed results are the estimates of the intercept β_0 and the slope β_1 in the hypothesized relationship

$$\text{PRIME}_t = \beta_0 + \beta_1 \text{ SUNSPOTS}_t + u_t \qquad \text{for } t=1,...,9.$$

This is an example of a simple linear regression. Note, however, that the outer product *SUNSPOTS*∘.⋆0 1 is merely a special case of a class of matrices of ascending powers of the single variable *SUNSPOTS*. That is, the expression

```
      SUNSPOTS∘.*0,⍳N
```

produces a matrix of 1+*N* columns, the columns containing *SUNSPOTS* raised to each of the powers 0 through *N* successively. For example, to accommodate two turning points in our estimated regression curve, we might want to fit *PRIME* to a cubic function of *SUNSPOTS*:

```
      SUNSPOTS∘.*0 1 2 3
  1    3    9   27
  1    6   36  216
  1    8   64  512
  1    8   64  512
  1    4   16   64
  1    7   49  343
  1    9   81  729
  1   11  121 1331
  1   13  169 2197
      PRIME PEGGYLEE SUNSPOTS∘.*0 1 2 3
0.1223741 ¯0.006414053521 0.002000414685 ¯9.392172483E¯5
```

This is an example of polynomial regression. We have specified the form of the hypothesized relationship, as well as its content, by the way in which we've constructed *PEGGYLEE*'s right argument. Note that we've introduced an implicit nonlinearity as well. *PEGGYLEE* is capable of estimating a nonlinear equation as long as that equation is what is called intrinsically linear, i.e. as long as it can be made linear by appropriate transformation.

What does *PEGGYLEE* look like? You might expect it to look like this:

```
      ∇Y PEGGYLEE X
[1]   (⌹(⍉X)+.×X)+.×(⍉X)+.×Y
      ∇
```

It could. Instead, however, line 1 is this expression:

```
[1]   Y⌹X
```

If your reaction to this line is "Is that all there is?", then you might realize why *PEGGYLEE* was selected as the name of the function.*

As is true of the other primitive functions we've seen, dyadic use of ⌹ is a generalized extension of its monadic use. The dyadic function represented by the domino symbol is called **matrix-divide**, and can be considered the matrix extension to scalar division. For example, in the simple scalar expression

```
      12÷4
3
```

the result 3 is that number which when *multiplied* by 4 (the right argument to the primitive function ÷) produces 12 (the left argument to ÷). Matrix-divide applies to matrices in the same manner. The expression *Y*⌹*X* produces a result matrix *Z* which, when **matrix-multiplied** by *X* (the right argument to the primitive function ⌹) produces the matrix *Y* (the left argument to ⌹). In particular, the matrix equation

$$\hat{y} = X\hat{\beta}$$

can be solved for β using the APL expression *Y*⌹*X*. Clearly, matrix-divide is ideally suited to econometric computation, since it can be used to solve a system of linear equations such as that given in (*) above.

To see that monadic ⌹ is indeed a special case of its more general dyadic counterpart, note that:

⌹*X* ↔ *I*⌹*X*

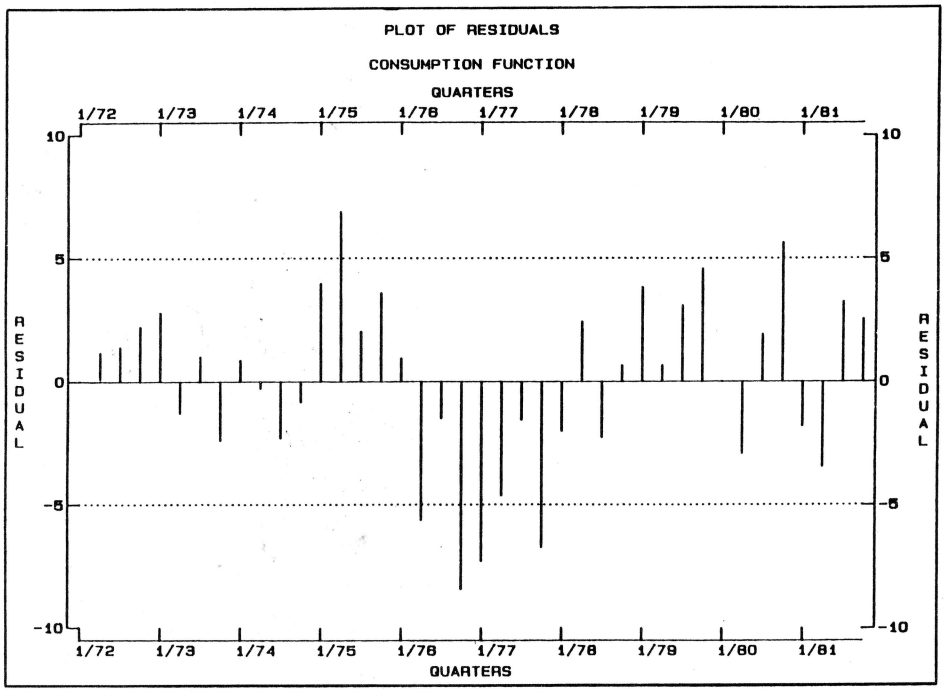
where *I* is the identity matrix of the same size as *X*.

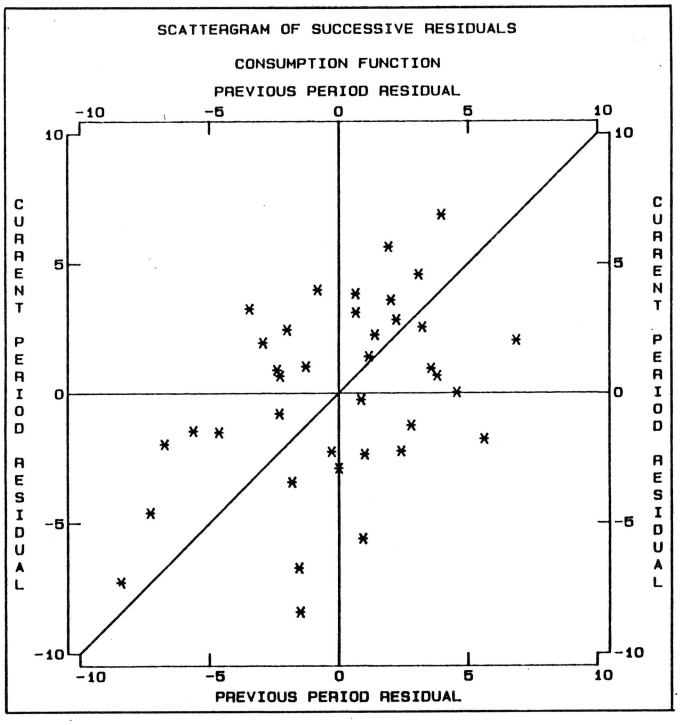
---

*Peggy Lee was a popular recording artist during the late 1950s and the 1960s. One of her more successful recordings was a song entitled "Is That All There Is?".

You'll notice that the output produced by *PEGGYLEE* consists solely of a non-formatted display of estimated coefficients. Don't get the impression that this is all that APL, or regression utilities written in APL, are capable of producing. The canned packages offered by most APL vendors generate comprehensive estimation results and diagnostic statistics, displayed in neat tabular arrangements; typical is the ordinary least squares output illustrated on the next page. Much public econometric software is also linked to a sophisticated graphics capability, enabling the production of visually appealing plots of publication quality. Two examples follow the sample regression output.

```
* REGRESS COMMAND: REGRESS CONS,2 0 4 0 0 ALMON INCOME

CONS  =  C0 + (C1 × 2 0 4 0 0 ALMON INCOME)

TIMEFRAME: QUARTERLY, DATED 1 73 TO 4 81

MEAN OF DEPENDENT VARIABLE  CONS          658.111

  VARIABLE                    MEAN      COEFFICIENT     STD. ERROR      T-VALUE
CONSTANT TERM                                1.531         15.536        0.099
INCOME                      4183.950         1.300          0.319        4.076
      (-1)                  2079.959        ¯4.797          1.605       ¯2.988
      (-2)                  1266.433         4.102          1.600        2.563


COEFFICIENT OF DETERMINATION R-SQUARED.....          0.9828
CORRECTED R-SQUARED (RBAR-SQUARED).........          0.9812
F-STATISTIC FOR SIGNIFICANCE OF REGRESSION( 3, 32)  608.3491
STANDARD ERROR OF THE ESTIMATE.............          7.6359
DURBIN-WATSON STATISTIC....................          0.5134
COEFFICIENT OF VARIATION (AT THE MEAN OF Y)..(o/o)   1.1603


DISTRIBUTED LAG INTERPRETATION -- LAGGED VARIABLE INCOME
 LAG  COEFFICIENT   STD. ERROR      T-VALUE
  0        0.615        0.101        6.075   |      +                    T  o  T
  1       ¯0.157        0.051       ¯3.071   |      +  T o T
  2       ¯0.073        0.089       ¯0.821   |T  o  T
  3       ¯0.075        0.051       ¯1.471   | T o T +
  4        0.151        0.103        1.469   |      +T  o   T
SUM        0.775        0.019       41.568

CONS  =      1.5310821085 + (    0.6146257823 × INCOME( 0))
       + (   ¯0.1568804708 × INCOME(-1)) + (   ¯0.0729767826 × INCOME(-2))
       + (   ¯0.0749459779 × INCOME(-3)) + (    0.1509728850 × INCOME(-4))
```


PLOT OF RESIDUALS
CONSUMPTION FUNCTION
QUARTERS
1/72
1/73
1/74
1/75
1/76
1/77
1/78
1/79
1/80
1/81
10
5
0
-5
-10
RESIDUAL
QUARTERS


SCATTERGRAM OF SUCCESSIVE RESIDUALS
CONSUMPTION FUNCTION
PREVIOUS PERIOD RESIDUAL
-10
-5
0
5
10
CURRENT PERIOD RESIDUAL
PREVIOUS PERIOD RESIDUAL

The APL expressions required to generate such full regression output employ the primitive functions we've seen. The estimated coefficients are calculated according to:

```
BETA←Y⌹X
```

(Recall that X must contain a column of 1s if a constant term is to be estimated.) The fitted y-values $\hat{y}$ are then given by $X\hat{\beta}$, or:

```
YHAT←X+.×BETA
```

and the residuals $\varepsilon = y - \hat{y}$ by:

```
E←Y-YHAT
```

The sum of squared residuals is:

```
SSR←+/E×E        (or SSR←E+.×E)
```

The degrees of freedom for the estimated regression are the number of observations minus the number of estimated parameters, namely:

```
N←1↑⍴X
K←1↓⍴X          (or K←⍴BETA)
DF←N-K
```

The ratio of the sum of squared residuals to the degrees of freedom produces the residual mean square $\hat{\sigma}^2$. The standard error of the estimate $\hat{\sigma}$ is then the square root of this value:

```
SIGMA←(SSR÷DF)*.5
```

The ratio of this standard error to the mean of the dependent variable produces the unit-free measure known as the coefficient of variation:

```
YBAR←(+/Y)÷N
CV←SIGMA÷YBAR
```

or, in percentage terms:

```
CV←100×SIGMA÷YBAR
```

The variance-covariance matrix of the estimated coefficients is $\hat{\sigma}^2 (X'X)^{-1}$, or:

```
VCV←(SIGMA*2)×⌹(⌹X)+.×X
```

It is the diagonal elements of this matrix which are the variances of the coefficient estimates. The square roots of these diagonal elements therefore represent the standard errors of the coefficients:

```
SE←(DIAG VCV)*.5
```

There does exist an APL primitive function which can be used to select the diagonal elements of a matrix. Since the function is one which we have not discussed here, assume we have available instead a monadic defined function *DIAG* (of explicit result) which serves the same purpose.

The t-statistics, used for testing for significant differences from zero of the estimated coefficients, are then simply the ratio of the coefficients to their standard errors:

```
T←BETA÷SE
```

The coefficient of determination R^2 measures the proportion of the variation in **Y** explained by the estimated equation; this is equivalent to:

$$R^2 = 1 - \frac{\varepsilon'\varepsilon}{y'y - n\bar{y}^2}$$

or, in APL, to:

```
R2←1-SSR÷(+/Y×Y)-N×YBAR*2
```

A slight adjustment for degrees of freedom produces corrected R^2, or $\bar{R}^2$:

```
RBAR2←1-((N-1)÷DF)×1-R2
```

A test of the significance of the overall relationship (excluding the intercept) can be performed using an F-statistic. F[(k-1),(n-k)] is calculated based on R^2 according to:

```
F←(R2÷K-1)÷(1-R2)÷DF
```

or

```
F←(DF×R2)÷(K-1)×1-R2
```

and its two appropriate degrees of freedom by:

```
FDF←(K-1),DF
```

A final statistic useful in testing for the existence of first-order serial correlation is the Durbin-Watson d-statistic, calculated from the residuals ϵ according to:

$$d = \frac{\sum_{t=2}^{n} (\varepsilon_t - \varepsilon_{t-1})^2}{\sum_{t=1}^{n} \varepsilon_t^2}$$

We saw earlier a suggestion that the take and drop functions might be useful here:

```
D←(+/((1↓E)-¯1↓E)*2)÷SSR
```

If the regression utility *PEGGYLEE* is to be capable of generating all of these equation statistics, then, it might look like this:

```
      ∇Y PEGGYLEE X
[1]   SSR←+/E×E←Y-YHAT←X+.×BETA←Y⌹X
[2]   N←1↑⍴X
[3]   DF←N-K←1↓⍴X
[4]   SIGMA←(SSR÷DF)*.5
[5]   CV←100×SIGMA÷YBAR←(+/Y)÷N
[6]   VCV←(SIGMA*2)×⌹(⌹X)+.×X
[7]   T←BETA÷SE←(DIAG VCV)*.5
[8]   RBAR2←1-((N-1)÷DF)×1-R2←1-SSR÷(+/Y×Y)-N×YBAR*2
[9]   F←(DF×R2)÷(K-1)×1-R2
[10]  FDF←(K-1),DF
[11]  D←(+/((1↓E)-¯1↓E)*2)÷SSR
      ∇
```

Relatively simple expressions such as these, involving no explicit iteration, are what actually constitute a canned APL regression program. Packages offered by various APL vendors might vary in the format of their printed results, but the numerical calculations performed are basically those of *PEGGYLEE*.

But we decided some time ago that we weren't really interested in developing our own software, but rather in using that offered by a supplier of packaged APL software. How can the power and flexibility of APL help us in this situation, when all the programming has already been completed? We saw one example of the answer to that question earlier: APL's array manipulation capabilities allow us to define the form and content of the hypothesized relationship, including implicit nonlinearities, by the way in which we construct the argument arrays.

As a further illustration, suppose that the defined function *PEGGYLEE* has been written in such a way that the right argument *X* can be either a matrix of variables or a vector containing a single variable, and that the column of 1s in *X* (representing the optional constant term) need not be included by the user. Most available software does operate in this manner.

Then the statement:

```
Y PEGGYLEE X
```

specifies and initiates a simple linear regression if *X* is a vector, and a multiple linear regression if *X* is a matrix of more than one column. If the columns of a matrix *X* are constructed in such a way (using ∘.*, for example) that they are ascending powers of a single variable, then *Y PEGGYLEE X* specifies and initiates the polynomial regression we saw earlier. Numerous other specifications are possible. Reciprocal regression, in which Y is a function of 1/X, is specified by:

```
Y PEGGYLEE ÷X
```

A semi-log specification is accomplished according to:

```
Y PEGGYLEE ⊛X
```

or:

```
(⊛Y) PEGGYLEE X
```

and a double-log, or log-log, specification according to:

```
(⊛Y) PEGGYLEE ⊛X
```

The incorporation of additional explanatory variables into a relationship is accomplished by simply catenating new columns onto the *X* matrix. For example, suppose we wish to incorporate a measure of consumption as a right-hand side variable, and that the measure is to be the sum of consumer expenditures on non-durable goods and those on services. This consumption measure could be catenated onto an existing matrix *X* of explanatory variables using the statement

```
X←X,GOODS+SERVICES
```

A new variable can be scaled, or deflated, using the dyadic function ÷:

```
X←X,PRICES÷DEFLATOR
```

To incorporate a time trend, we can catenate the numbers 1 through n:

```
X←X,⍳N
```

Instead, the time variable can be represented by the n years which constitute the post-World War II era:

```
X←X,1945+⍳N
```

Or, the time variable could be centred around some key base period:

```
      ¯8+⍳15
¯7 ¯6 ¯5 ¯4 ¯3 ¯2 ¯1 0 1 2 3 4 5 6 7
      X←X,(-⌈N÷2)+⍳N
```

Additional explanatory variables can be dummy variables as well. Recall that the function reshape (dyadic ⍴) recycles the elements of its right argument if there are fewer of them than the number specified in its left argument. This means that the expressions

```
      X←X,N⍴1 0 0 0
      X←X,N⍴0 1 0 0
      X←X,N⍴0 0 1 0
```

suffice to incorporate three quarterly seasonal dummies. Dummy variables can also be calculated based upon some condition. For example, we may need to construct a dummy variable which has the value 0 before the Arab oil embargo of 1973, and 1 from that year onwards. If a vector *TIME* contains the 15 years 1967 through 1981, say, then the required oil embargo dummy can be generated as *TIME*≥1973:

```
      TIME←1966+⍳15
      TIME
1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981
      TIME≥1973
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
      X←X,TIME≥1973
```

Such dummy variables, of course, affect only the intercept of the estimated equation. To allow a slope coefficient to change in response to a particular event or circumstance, we need an interactive dummy. Interactive dummies are specified similarly to regular dummies. The slope coefficient on a variable *IMPORTS*, for example, could be allowed to change at the time of the oil embargo were we to specify this interactive dummy:

```
      X←X,IMPORTS×TIME≥1973
```

All of the simple illustrations above involve relationships in the current period only. No lag or lead times are applied to any variable. But, since in practice the effect of an explanatory variable is often not felt until some time has elapsed, we need to be able to incorporate lagged variables into relationships as well. Consider for the last time the variables representing the prime lending rate and the sunspot count:

```
      PRIME
0.115 0.125 0.14 0.145 0.13 0.16 0.165 0.17 0.17
      SUNSPOTS
3 6 8 8 4 7 9 11 13
```

Hypothesize, if you will, that the prime rate is related not only to the sunspot count in the current period, but also to that in each of the previous three periods. How are we to impart this information (this equation specification) to *PEGGYLEE*? We need to feed *PEGGYLEE* a right argument matrix containing four columns: the variable *SUNSPOTS* lagged 0, 1, 2 and 3 periods respectively. We can construct such a matrix this way:

```
      4 9⍴SUNSPOTS
3  6  8  8  4  7  9 11 13
3  6  8  8  4  7  9 11 13
3  6  8  8  4  7  9 11 13
3  6  8  8  4  7  9 11 13
      0 ¯1 ¯2 ¯3⌽4 9⍴SUNSPOTS
 3  6  8  8  4  7  9 11 13
13  3  6  8  8  4  7  9 11
11 13  3  6  8  8  4  7  9
 9 11 13  3  6  8  8  4  7
```

The nine elements of the vector *SUNSPOTS* are reshaped (and recycled) into a matrix of four rows and nine columns. The four rows are then rotated (across columns) 0, 1, 2 and 3 positions to the right respectively. But *PEGGYLEE* requires an *X* matrix whose columns, not rows, contain the independent variables:

```
      ⍉0 ¯1 ¯2 ¯3 ⌽4 9⍴SUNSPOTS
 3 13 11  9
 6  3 13 11
 8  6  3 13
 8  8  6  3
 4  8  8  6
 7  4  8  8
 9  7  4  8
11  9  7  4
13 11  9  7
```

Application of transpose produces the required *X* matrix. We can initiate the regression of the prime rate on the various lagged sunspot counts, then, using the statement:

```
PRIME PEGGYLEE ⍉0 ¯1 ¯2 ¯3⌽4 9⍴SUNSPOTS
```

Note that the results of this estimation would be incorrect, however. This is because the elements inside the triangle shown above are spurious and inappropriate, but have been accepted indiscriminately by *PEGGYLEE* as legitimate observations. We must remove the spurious "observations" before passing the data to *PEGGYLEE* via its right argument:

```
      3 0↓⌽0 ¯1 ¯2 ¯3⌽4 9⍴SUNSPOTS
 8  8  6  3
 4  8  8  6
 7  4  8  8
 9  7  4  8
11  9  7  4
13 11  9  7
```

The first 3 rows (and 0 columns), i.e. those which contained the illegitimate observations, are dropped. This matrix, then, is the proper right argument matrix required by *PEGGYLEE*:

```
(3↓PRIME) PEGGYLEE 3 0↓⌽0 ¯1 ¯2 ¯3⌽4 9⍴SUNSPOTS
```

Note that the dependent variable *PRIME* has been truncated accordingly. This loss of observations is a feature of any relationship involving lag, lead or differencing transformations. For example, to regress *PRIME* against first differences of *SUNSPOTS*, we would have to drop the first element of *PRIME*, since an element of *SUNSPOTS* has been lost through differencing:

```
(1↓PRIME) PEGGYLEE (1↓SUNSPOTS)-¯1↓SUNSPOTS
```

Clearly, APL's capabilities afford us the luxury of using a single ordinary least squares regression program to estimate a variety of specifications, including dynamic relationships and certain nonlinearities. We control the content and the form of the estimated equation by the way in which we construct the data arguments to this one program.

Appendix: Dealing with common regression pitfalls

Our discussion of APL applied to econometrics has been restricted so far to the use of ordinary least squares. If any of the assumptions fundamental to OLS is violated, certain corrective measures must be taken, and some alternative estimation technique employed. Let's consider quickly three of the more common regression pitfalls, and the application of APL to various forms of generalized least squares (GLS).

The first pitfall is **heteroscedasticity**. Recall that one of the assumptions underlying OLS is that the variance-covariance matrix Ω of the errors be scalar diagonal, i.e. that Ω have constant diagonal elements and zeros elsewhere. Since the diagonal elements of Ω are the variances of the errors, this assumption embodies what is referred to as the constant variance assumption. Errors which exhibit a constant variance are called homoscedastic; those which do not are called heteroscedastic.

One simple test of the constant variance assumption is provided by the **Spearman rank correlation coefficient**. This is a coefficient of simple correlation between the rank (not to be confused with the APL definition of rank) of the absolute values of the OLS errors and the rank of that particular explanatory variable x_j with which the variance of the errors might be associated. (The test requires that we hypothesize that the magnitude of the errors varies in accordance with the magnitude of one of the independent variables.) The OLS utility *PEGGYLEE* calculates a vector called *E* which contains the errors. The ranks of the absolute values of these errors are calculated using the upgrade function:

```
RANKERROR←⍋|E
```

Similarly, if the associated explanatory variable x_j is available as the vector *XJ*, its ranks are given by:

```
RANKXJ←⍋XJ
```

Our defined function *CORRELATE* from earlier can then be used to calculate the coefficient of simple correlation between the two rank vectors:

```
RANKERROR CORRELATE RANKXJ
```

This result is the Spearman rank correlation coefficient, and is subject to the same t-test as is any other coefficient of simple correlation.

An alternative test for the existence of heteroscedastic disturbances involves the application of the **Glejser regressions**. These regressions suggest the estimation of relationships of the form:

$$|\epsilon| = \alpha + \beta x_j^{\gamma}$$

where the values ϵ are the OLS errors and x_j is again that independent variable with which we suspect the error variance to be associated. The parameter γ is assigned systematic values between -2 and 2, say. The decision concerning the existence of heteroscedasticity then depends upon the statistical significance of the estimated coefficients $\hat{\alpha}$ and $\hat{\beta}$. The Glejser regressions are performed very simply in APL by the statement:

```
(|E) PEGGYLEE XJ*GAMMA
```

The parameter *GAMMA* could be first assigned the value ¯2, then ¯1.5, then ¯1, and so on. Note that the direct application of matrix-divide, i.e.:

```
(|E)⌹(XJ*GAMMA)∘.*0 1
```

would produce the same coefficient estimates, but would provide no additional statistics with which we could test their significance.

If the Spearman rank correlation coefficient or the Glejser regressions (or any other test, such as the Goldfeld-Quandt F-test) suggest the existence of a heteroscedastic error term, remedial action is necessary to ensure that the estimated coefficients are minimum variance estimates. The usual remedy is the application of generalized least squares. This involves specifying an alternative structure for the variance-covariance matrix Ω of the errors. Since it is the constant variance assumption of OLS that is in question, it is the diagonal elements of Ω (representing variances) that need to be specified. That is, OLS assumes that Ω looks like this:

$$\Omega = \sigma^2 I = \begin{bmatrix} \sigma^2 & & & & O \\ & \sigma^2 & & & \\ & & \sigma^2 & & \\ & & & \ddots & \\ O & & & & \sigma^2 \end{bmatrix}$$

In order to correct for heteroscedasticity, we need to respecify so that its diagonal elements are not all the same. This is very easily accomplished in APL.

For example, consider a vector *V*:

```
      V
6 8 9 11 14
```

We can construct a 5×5 matrix whose diagonal elements are the five elements of *V* as follows:

```
      V,5 5ρ0
 6  0  0  0  0  0
 8  0  0  0  0  0
 9  0  0  0  0  0
11  0  0  0  0  0
14  0  0  0  0  0
      5 5ρV,5 5ρ0
 6  0  0  0  0
 0  8  0  0  0
 0  0  9  0  0
 0  0  0 11  0
 0  0  0  0 14
```

The vector *V* is first catenated to a 5×5 matrix of zeros, so that the elements of *V* constitute the first column of a 5×6 matrix. This matrix is then reshaped into a 5×5 result.

There are in practice a number of ways to specify the diagonal elements of Ω so that GLS can be applied. One commonly used method is to place the square of each of the OLS errors along the diagonal. We can do this in APL as follows:

```
OMEGA←(N,N)ρ(E*2),(N,N)ρ0
```

using the variables *N* (number of observations) and *E* (OLS errors) produced by *PEGGYLEE*. If Glejser regressions are estimated and suggest heteroscedasticity (for a given γ), then we can use as the diagonal elements of Ω the squares of the fitted values from the estimated equation. We do so according to:

```
OMEGA←(N,N)ρ(YHAT*2),(N,N)ρ0
```

where *YHAT* is the vector of fitted y-values produced by *PEGGYLEE*.

Note that all of these calculations require numerical results from a previous OLS analysis. This poses no problem, since it is presumably an examination of OLS results that suggests the existence of heteroscedasticity in the first place.

A second common econometric pitfall is **autocorrelation**. The OLS assumption violated by autocorrelation again concerns the variance-covariance matrix of errors. In this instance it is the off-diagonal elements, not the diagonal elements, which cause the problem. The assumption that Ω's off-diagonal elements are zero reflects the desire that there be no covariance between one error and a different error, i.e. that the errors be independent of one another. This assumption is frequently violated when we're dealing with time series data, in which the error in the current period is often related to the error in the previous period, and possibly to others in earlier periods. If OLS is applied

to a relationship in which the errors are related in this way, the resulting estimated coefficients will again not be best in the minimum variance sense.

Conventional test statistics such as the Durbin-Watson d-statistic and Durbin's h-statistic are common means of testing for the presence of autocorrelated errors. The remedy for autocorrelation is the same as that for heteroscedasticity: application of some form of generalized least squares. (Note that autocorrelation adjustment techniques such as those proposed by Cochrane and Orcutt and by Hildreth and Lu are in effect special cases of GLS.)

What we have to do to correct for autocorrelation, then, is construct a matrix whose off-diagonal elements are non-zero. The most commonly used structure for such a matrix is this:

$$\Omega = \begin{bmatrix} 1 & \rho & \rho^2 & \cdots & \rho^{n-1} \\ \rho & 1 & \rho & \rho^2 & \cdot \\ \rho^2 & \rho & 1 & \rho & \cdot \\ \cdot & & & \cdot & \cdot \\ \cdot^{n-1} & & & \cdot & \cdot \\ \rho & & \cdots & \rho & 1 \end{bmatrix}$$

The parameter ρ is a number between -1 and 1 measuring the extent to which the error in the current period is related to the error in the previous period. The two diagonals on either side of the main diagonal of Ω consist of the constant ρ; these elements represent covariances between errors one period apart, and reflect the assumption that

$$\varepsilon_t = \rho \varepsilon_{t-1}$$

The next pair of diagonals, moving outwards from the main diagonal, consist of the constant ρ ; these elements represent covariances between errors two periods apart, and reflect the assumption that

$$\begin{aligned} \varepsilon_t &= \rho \varepsilon_{t-1} \\ &= \rho(\rho \varepsilon_{t-2}) \\ &= \rho^2 \varepsilon_{t-2} \end{aligned}$$

In general, the elements of those diagonals removed k diagonals from the main diagonal consist of the constant ρ^k.

There are a number of simple ways in which we can estimate ρ. The simplest is:

$$\hat{\rho} = 1 - \frac{d}{2}$$

where d is the Durbin-Watson d-statistic from a previous OLS analysis. In APL:

```
RHO←1-D÷2
```

where *D* is as produced by *PEGGYLEE*. Another method is to estimate the autoregressive relationship:

$$\varepsilon_t = \rho\,\varepsilon_{t-1} + \varepsilon'_t$$

via OLS. The resulting estimated slope coefficient (there is no intercept) is then the required estimate of ρ. In APL, this is simply:

```
RHO←(1↓E)⌹¯1↓E
```

(Recall that $1\downarrow E$ represents ε_t for t=2,...,n, and $\bar{}1\downarrow E$ represents ε_{t-1} for the same subscripts t.)

Given a value for $\hat{\rho}$, we can construct Ω using an outer product. Suppose, for example, that n=5. Then:

```
      N←5
      ⍳N
1 2 3 4 5
      (⍳N)∘.-⍳N
 0 ¯1 ¯2 ¯3 ¯4
 1  0 ¯1 ¯2 ¯3
 2  1  0 ¯1 ¯2
 3  2  1  0 ¯1
 4  3  2  1  0
      |(⍳N)∘.-⍳N
0 1 2 3 4
1 0 1 2 3
2 1 0 1 2
3 2 1 0 1
4 3 2 1 0
```

The outer product ∘.- produces a 5×5 matrix containing simple differences between all possible pairs of elements of the vector ⍳5. The absolute value of this result matrix is then the matrix of powers to which $\hat{\rho}$ must be raised in the construction of Ω:

```
OMEGA←RHO*|(⍳N)∘.-⍳N
```

The matrix *OMEGA* can then be provided as an input to a canned GLS program, and the original relationship estimated anew.

The usefulness of the estimated parameter $\hat{\rho}$ does not end here. It is required again if the estimated equation is to be used for prediction. Prediction based on an equation estimated via OLS is performed by calculating

$$\mathbf{f} = x_p \hat{\beta}$$

where x_p is a matrix of independent variables over the p prediction periods. In APL this is simply:

```
F←XP+.×BETA
```

But when an equation has been estimated using GLS in an attempt to adjust for the effects of autocorrelation, a correction factor must be added to any predicted values. This correction factor reflects the fact that the original OLS errors were related to one another. For the first prediction period, the additive correction is an amount

$$\hat{\rho}\varepsilon_n$$

where ϵ is the GLS residual from the last estimation period. The correction for the second prediction period is

$$\hat{\rho}^2 \varepsilon_n = \hat{\rho}\hat{\rho}\varepsilon_n$$

and so on. In general, the additive correction for the k-th prediction period is the amount

$$\hat{\rho}^k \varepsilon_n$$

Suppose the vector *F* has been calculated as above, and contains uncorrected predictions for p periods, and that the vector *E* contains the n GLS errors. The p corrected predictions can then be calculated in APL using:

```
F←F+(¯1↑E)×RHO*⍳P
```

Note that these corrected predictions could also be calculated in a less obvious manner, using the scan operator:

```
F←F+(¯1↑E)××\P⍴RHO
```

The third econometric pitfall commonly encountered is **multicollinearity**. Recall that this is a violation of an OLS assumption concerning not the error term, but rather the matrix **X** of independent variables. The assumption is called the full rank condition, and requires that the independent variables be just that, i.e. independent of one another. If one of the x variables is an exact linear combination of one or more other x variables, then the matrix **X'X** cannot be inverted, and the regression coefficients cannot be

estimated. It is when one of the x variables is an approximate linear combination of one or more other x variables that multicollinearity problems arise in practice. Under such circumstances, the calculated t-statistics on seemingly important variables tend to be low (the result of the inflated standard errors), and certain of the estimated coefficients might have the opposite sign to that expected.

Remedies for multicollinearity are numerous. Imposition of a distributed lag structure on the estimated coefficients is a popular, but relatively complicated method. There are easier ways to deal with the problem. One is a simple two-stage procedure. Having tentatively identified, possibly using the correlation matrix, those x variables which are linearly dependent, we can regress each of these variables in turn against the others, and test the significance of each regression using an F-test. We select that relationship which appears strongest (the "best fit", if there is such a thing). The fitted values, or $\hat{y}$'s, from this regression can then be used in place of the offending (linearly dependent) x variables in the original equation. To perform such a substitution in APL, we need simply construct a new matrix *X* of explanatory variables. *X* no longer includes those variables which manifest linear dependence, but instead has catenated to it a vector *YHAT*. *YHAT* contains the fitted values generated by *PEGGYLEE* in its estimation of the strongest relationship among the original x variables.

Note that this approach is equivalent to incorporating a new variable which is a weighted sum of a number of other variables. A variation on this theme, also useful in dealing with problems of multicollinearity, is the calculation of moving averages of previous values of a single variable. When a number of different lags of the same variable are included as independent variables in an equation, some degree of linear dependence between these lagged values is likely. Substitution of a moving average of the original variable in place of its various lagged values serves to remove the linear dependence.

For calculating moving averages in APL, the outer product is again a convenient tool. Consider a vector *UR* representing a proxy for the unemployment rate (%) over 14 quarters:

```
      UR
6 5.9 5.9 5.7 5.8 6 6.3 7.3 7.7 7.4 7.4 7.4 7.4 8.4
```

Suppose we wish to calculate a 4-quarter simple moving average of *UR*. It would be convenient if we could somehow isolate elements 1 through 4 of *UR*, then elements 2 through 5, and so on, until we had used all 14 elements. Recall that the outer product ∘.+ can be used to produce a table of ascending indices:

```
      0 1 2 3∘.+⍳11
 1  2  3  4  5  6  7  8  9 10 11
 2  3  4  5  6  7  8  9 10 11 12
 3  4  5  6  7  8  9 10 11 12 13
 4  5  6  7  8  9 10 11 12 13 14
```

If we use this matrix to index into the vector *UR*, the result is a matrix whose first column contains elements 1 through 4 of *UR*, whose second column contains elements 2 through 5 of *UR*, and so on:

```
      UR[0 1 2 3∘.+⍳11]
6   5.9 5.9 5.7 5.8 6   6.3 7.3 7.7 7.4 7.4
5.9 5.9 5.7 5.8 6   6.3 7.3 7.7 7.4 7.4 7.4
5.9 5.7 5.8 6   6.3 7.3 7.7 7.4 7.4 7.4 7.4
5.7 5.8 6   6.3 7.3 7.7 7.4 7.4 7.4 7.4 8.4
```

Summing along the rows, i.e. down each column, and dividing by 4, then yields the required 4-quarter moving average:

```
      (+/[1] UR[0 1 2 3∘.+⍳11])÷4
5.875 5.825 5.85 5.95 6.35 6.825 7.175 7.45 7.475 7.4 7.65
```

In general, to calculate an *L*-period simple moving average of an *N*-element vector *X*, we use the expression:

```
      (+/[1] X[(0,⍳L-1)∘.+⍳1+N-L])÷L
```

In the case of the vector *UR*, this expression is:

```
      N←⍴UR
      L←4
      (+/[1] UR[(0,⍳L-1)∘.+⍳1+N-L])÷L
5.875 5.825 5.85 5.95 6.35 6.825 7.175 7.45 7.475 7.4 7.65
```

Note that a simple moving average is a special case of a weighted moving average. It is a weighted moving average in which all weights are equal. What we have done in our calculations above is equivalent to selecting every four successive elements of *UR*, multiplying the four elements by an amount .25, and summing them. You might recall that performing element-by-element multiplication, then summing the resulting products, is exactly what the inner product +.× accomplishes. That is, our simple moving average could also be calculated as a weighted moving average according to:

```
      .25 .25 .25 .25+.×UR[0 1 2 3∘.+⍳11]
5.875 5.825 5.85 5.95 6.35 6.825 7.175 7.45 7.475 7.4 7.65
```

The four elements in the first column of the matrix to the right of the derived function +.× (i.e. the first four elements of *UR*) are multiplied by the four values of .25, and the four resulting products summed to produce 5.875. Because there are 11 columns in +.×'s right argument matrix, there are 11 elements in the result.

A weighted moving average in which not all weights are equal is calculated in the same way. For example, suppose we wish to assign the two central periods of our 4-quarter moving average 1.5 times the weight assigned to the first and last periods. We could do so as follows:

```
      WEIGHTS←.2 .3 .3 .2
```

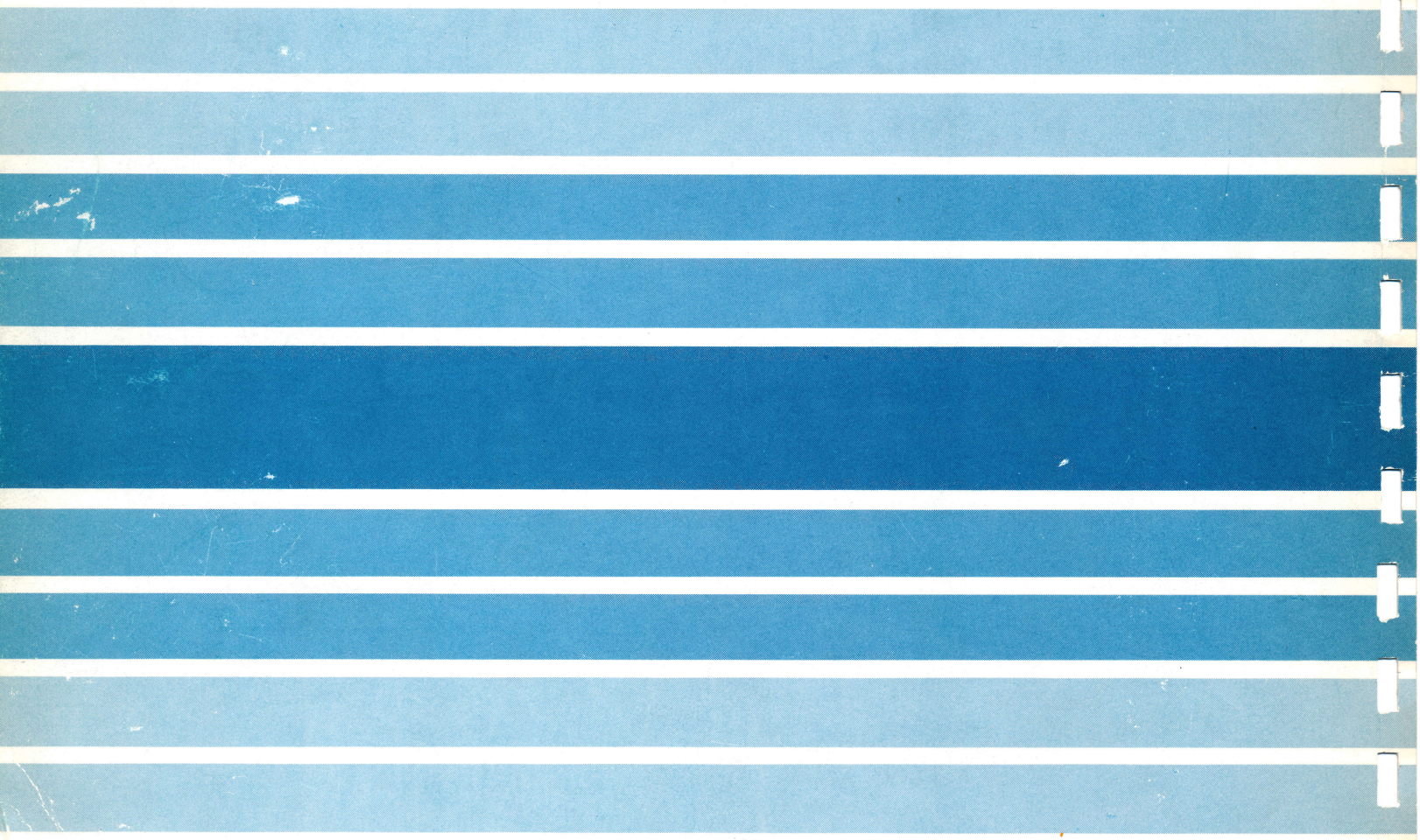
The weighted moving average of the unemployment rate would then be given by:

```
      WEIGHTS+.×UR[0 1 2 3∘.+⍳11]
5.88 5.82 5.83 5.94 6.31 6.82 7.24 7.47 7.46 7.4 7.6
```

The proliferation of moving average calculations in various forecasting algorithms makes expressions such as those given above extremely useful. Weighted moving averages of different lengths are the basis of classical decomposition methods, and therefore of most common seasonal adjustment algorithms, including the Census Method II, X-11.

* * *

There is far more to APL than what we've discussed here. The objective of this tutorial has not been to turn you instantly into an APL programmer, but rather to give you some idea of how you can use APL as a tool for practical statistical and econometric analysis. Mark Twain once commented that there are three kinds of lies: lies, damned lies, and statistics. While APL can't improve your veracity, it can certainly ease the computational burden of everyday statistics.



Printed in Canada